ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

NGUYỄN THỊ YÊN

CÁC KỸ THUẬT TRONG KIỂM THỬ DÒNG DỮ LIỆU TĨNH

LUẬN VĂN THẠC SĨ KỸ THUẬT PHẦN MỀM

Hà Nội - 2016

ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

NGUYỄN THỊ YÊN

CÁC KỸ THUẬT TRONG KIỂM THỬ DÒNG DỮ LIỆU TĨNH

Ngành: Công nghệ thông tin
Chuyên ngành: Kỹ thuật phần mềm
Mã số: 6048103

LUẬN VĂN THẠC SĨ KỸ THUẬT PHẦN MỀM

NGƯỜI HƯỚNG DẪN KHOA HỌC: TS. Đặng Văn Hưng

Hà Nội - 2016

LỜI CAM ĐOAN

Tôi xin cam đoan:

Những kết quả nghiên cứu được trình bày trong luận văn là hoàn toàn trung thực, của tôi, không vi phạm bất cứ điều gì trong luật sở hữu trí tuệ và pháp luật Việt Nam. Nếu sai, tôi hoàn toàn chịu trách nhiệm trước pháp luật.

TÁC GIẢ LUẬN VĂN

Nguyễn Thị Yên

MỤC LỤC

Trang

DANH MỤC CÁC KÝ HIỆU VIẾT TẮT

TT	Viết tắt	Đầy đủ	Diễn giải
1	TBFV	Testing - Based Formal Verification	Kỹ thuật chứng minh hình thức dựa trên kiểm thử
2	RUP	Rational Unified Process	Qui trình phát triển phần mềm
3	DU-path	Definition Use path	Đường dẫn định nghĩa sử dụng
4	FSF	Functional Scenario Form	Hình thức kịch bản chức năng
5	SOFL	Structured Object-Oriented Formal Language	Ngôn ngữ hình thức hướng đối tượng cấu trúc

DANH MỤC CÁC HÌNH

Trang

DANH MỤC CÁC BẢNG

Trang

MỞ ĐẦU

Chúng ta đã và đang chứng kiến sự tăng trưởng đáng kinh ngạc của ngành công nghiệp phần mềm trong vài thập kỷ qua. Nếu như trước đây phần mềm máy tính chỉ được sử dụng để tính toán khoa học kỹ thuật và xử lý dữ liệu thì ngày nay nó đã được ứng dụng vào mọi mặt của đời sống hàng ngày của con người, từ các ứng dụng nhỏ để điều khiển các thiết bị dùng trong gia đình như các thiết bị nghe nhìn, điện thoại, máy giặt, lò vi sóng, nồi cơm điện, đến các ứng dụng lớn hơn như trợ giúp điều khiển các phương tiện và hệ thống giao thông, trả tiền cho các hóa đơn, quản lý và thanh toán về tài chính, ... Vì thế con người ngày càng phụ thuộc chặt chẽ vào các sản phẩm phần mềm và do vậy đòi hỏi về chất lượng của các sản phẩm phần mềm tăng, giá thành hạ, sử dụng dễ dàng, an toàn và tin cậy được. Kiểm thử có phương pháp là một hoạt động không thể thiếu trong quy trình sản xuất phần mềm để đảm bảo các yếu tố chất lượng nêu trên của các sản phẩm phần mềm.

Theo thống kê thì việc kiểm thử tiêu tốn khoảng 50% thời gian và hơn 50% giá thành của các dự án phát triển phần mềm. Tăng năng suất kiểm thử là một nhu cầu thiết yếu để tăng chất lượng phần mềm.

Từ những lý do trên nên em đã chọn đề tài: ***"Các kỹ thuật trong kiểm thử dòng dữ liệu tĩnh"***.

Mục tiêu của đề tài: Nghiên cứu Tổng quan về kiểm thử phần mềm để nắm những kiến thức cơ bản phục vụ cho các nghiên cứu tiếp theo. Sau đó nghiên cứu Tổng quan về các phương pháp kiểm thử phần mềm và kiểm thử dòng dữ liệu tĩnh. Tiếp theo nghiên cứu ứng dụng Logic Hoare trong kiểm thử phần mềm, cụ thể: nghiên cứu kỹ thuật kết hợp Logic Hoare với kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu và áp dụng kỹ thuật kết hợp này vào kiểm thử một đoạn chương trình.

Cấu trúc của luận văn được chia thành 3 chương cụ thể như sau:

Chương 1: Tổng quan về Kiểm thử phần mềm và kiểm thử phần mềm tĩnh

Trình bày những khái niệm cơ bản liên quan đến lĩnh vực Kiểm thử phần mềm như khái niệm kiểm thử phần mềm, vai trò của Kiểm thử phần mềm, các mức độ trong kiểm thử phần mềm... Đồng thời cũng trình bày khái quát về kiểm thử phần mềm tĩnh.

Chương 2: Phương pháp kiểm thử dòng dữ liệu tĩnh trong kiểm thử phần mềm

Trình bày cách phân loại kiểm thử dòng dữ liệu tĩnh và trình bày sơ lược một số kỹ thuật kiểm thử dòng dữ liệu tĩnh.

Chương 3: Ứng dụng Logic Hoare trong kiểm thử phần mềm

Trình bày tổng quan về Logic Hoare và kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu. Sau đó trình bày kỹ thuật kiểm thử phần mềm kết hợp giữa Logic Hoare và kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu để nâng cao hiệu quả cho kỹ thuật kiểm thử phần mềm dựa trên kịch bản dòng dữ liệu. Cuối cùng ứng dụng phương pháp kết hợp vào kiểm thử một đoạn chương trình.

Để hoàn thành được luận văn, em xin được gửi lời cảm ơn tới các thầy cô trong Khoa Công nghệ thông tin - Trường Đại học Công nghệ đã tận tình giảng dạy, cung cấp nguồn kiến thức quý giá trong suốt quá trình học tập. Đặc biệt em xin chân thành cảm ơn thầy giáo TS. Đặng Văn Hưng đã tận tình hướng dẫn, góp ý, tạo điều kiện cho em hoàn thành luận văn này.

Chương 1
TỔNG QUAN VỀ KIỂM THỬ PHẦN MỀM VÀ KIỂM THỬ TĨNH

Trong chương này, tác giả luận văn sẽ trình bày những kiến thức cơ bản liên quan đến Kiểm thử phần mềm như định nghĩa kiểm thử phần mềm, các mức kiểm thử phần mềm… Đồng thời tác giả cũng trình bày khái quát về Kiểm thử tĩnh, trong đó tác giả trình bày sơ lược các kỹ thuật kiểm thử phần mềm tĩnh.

1.1. Khái quát về Kiểm thử phần mềm

1.1.1. Định nghĩa về Kiểm thử phần mềm

Theo tài liệu [1], Kiểm thử phần mềm liên quan đến các khái niệm: lỗi (Error), sai (Fault), thất bại (Failure) và sự cố (Incident) [1]. Có hai mục đích chính của một phép thử: tìm thất bại hoặc chứng tỏ việc tiến hành của phần mềm là đúng đắn.

Vai trò của Kiểm thử phần mềm

Kiểm thử phần mềm đóng vai trò quan trọng trong việc đánh giá và thu được chất lượng cao của sản phẩm phần mềm trong quá trình phát triển. Thông qua chu trình "*kiểm thử - tìm lỗi - sửa lỗi"*, chúng ta hy vọng chất lượng của sản phẩm phần mềm sẽ được cải tiến. Mặt khác, thông qua việc tiến hành kiểm thử mức hệ thống trước khi cho lưu hành sản phẩm, chúng ta biết được sản phẩm của chúng ta tốt ở mức nào. Vì thế, nhiều tác giả đã mô tả việc kiểm thử phần mềm là một quy trình kiểm chứng để đánh giá và tăng cường chất lượng của sản phẩm phần mềm. Quy trình này gồm hai công việc chính là phân tích tĩnh [1] và phân tích động [1]. Bằng việc phân tích tĩnh và động, người kiểm thử có thể phát hiện nhiều lỗi nhất có thể để chúng có thể được sửa ở giai đoạn sớm nhất trong quá trình phát triển phần mềm. Phân tích tĩnh và động là hai kỹ thuật bổ sung cho nhau và cần được làm lặp đi lặp lại nhiều trong quá trình kiểm thử.

Định nghĩa kiểm thử phần mềm [1]

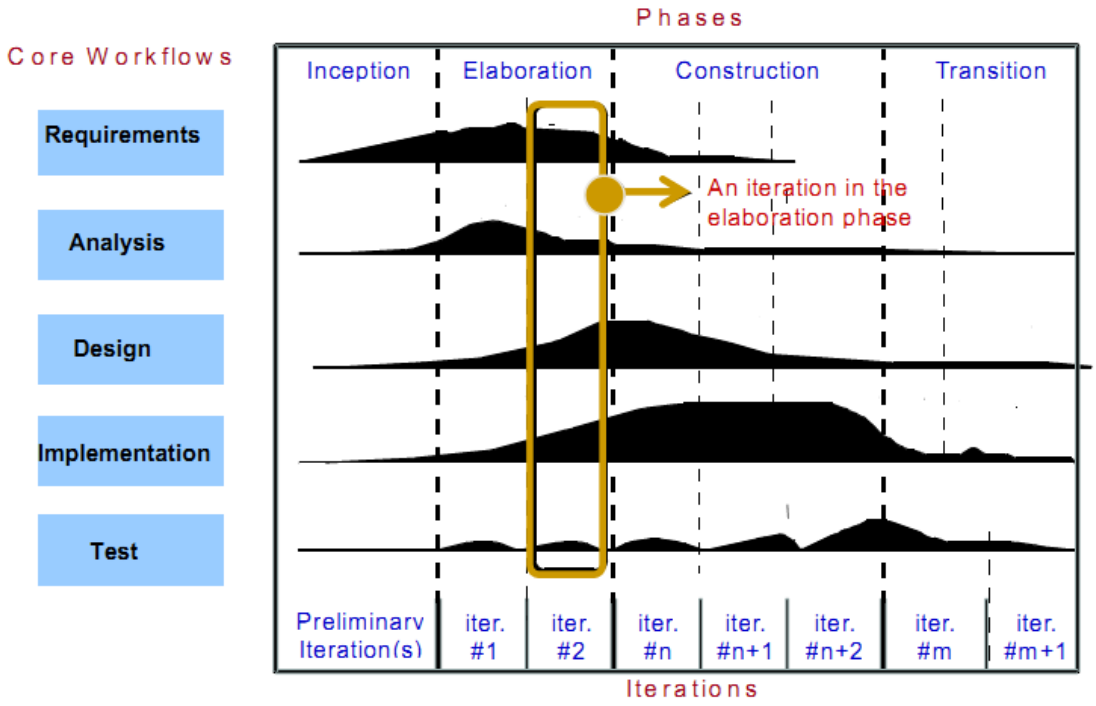
Kiểm thử phần mềm là qui trình nhằm đảm bảo chất lượng phần mềm. Kiểm thử phần mềm hướng tới việc chứng minh phần mềm không có lỗi.

Mục đích của kiểm thử phần mềm là phát hiện lỗi càng sớm càng tốt, và đảm bảo rằng những lỗi này phải được sửa. Lỗi được hiểu là phần mềm không hoạt động đúng như đặc tả của nó.

Trước khi đi vào trình bày các vấn đề khác liên quan đến kiểm thử phần mềm, trước tiên luận văn sẽ trình bày qui trình phát triển một phần mềm để có

cái nhìn tổng thể hơn về kiểm thử phần mềm gắn kết với các giai đoạn phát triển phần mềm.

1.1.2. Qui trình phát triển phần mềm RUP [1]



Hình 1.1: Qui trình phát triển phần mềm RUP

Chu kỳ phần mềm được tính từ lúc có yêu cầu (mới hoặc nâng cấp) đến lúc phần mềm đáp ứng đúng yêu cầu được phân phối.

Trong mỗi chu kỳ, người ta tiến hành nhiều công đoạn: Khởi động, chi tiết hóa, hiện thực, và chuyển giao.

Mỗi công đoạn thường được thực hiện theo cơ chế lặp nhiều lần để kết quả ngày càng hoàn hảo hơn.

Trong từng bước lặp, chúng ta thường thực hiện nhiều workflow đồng thời (để tận dụng nguồn nhân lực hiệu quả nhất): Nắm bắt yêu cầu, phân tích chức năng, thiết kế, hiện thực và kiểm thử.

Sau mỗi lần lặp thực hiện một công việc nào đó, chúng ta phải tạo ra kết quả (artifacts), kết quả của bước/công việc này là dữ liệu đầu vào của bước/công việc khác. Nếu thông tin không tốt sẽ ảnh hưởng nghiêm trọng đến kết quả của các bước/hoạt động sau đó.

Một số vấn đề thường gặp trong phát triển phần mềm:

- Tính toán không đúng, hiệu chỉnh sai dữ liệu.
- Trộn dữ liệu không đúng.
- Tìm kiếm dữ liệu sai yêu cầu.

- Xử lý sai mối quan hệ giữa các dữ liệu.
- Coding/hiện thực sai các qui luật nghiệp vụ.
- Hiệu suất của phần mềm còn thấp.
- Kết quả hoặc hiệu suất phần mềm không tin cậy.
- Hỗ trợ chưa đủ các nhu cầu nghiệp vụ.
- Giao tiếp với hệ thống khác chưa đúng hay chưa đủ.
- Kiểm soát an ninh phần mềm chưa đủ.

Các mục tiêu chính của Kiểm thử phần mềm [1]:

- Phát hiện càng nhiều lỗi càng tốt trong thời gian kiểm thử xác định trước.
- Chứng minh rằng sản phẩm phần mềm phù hợp với các đặc tả yêu cầu của nó.
- Xác thực chất lượng kiểm thử phần mềm đã dùng chi phí và nỗ lực tối thiểu.
- Tạo các ca kiểm thử chất lượng cao, thực hiện kiểm thử hiệu quả và tạo ra các báo cáo vấn đề đúng và hữu dụng.

Kiểm thử phần mềm là một thành phần trong lĩnh vực rộng hơn, đó là Verification & Validation (V&V), tạm dịch là Thanh kiểm tra và Kiểm định phần mềm.

Thanh kiểm tra phần mềm là qui trình xác định xem sản phẩm của một công đoạn trong qui trình phát triển phần mềm có thỏa mãn các yêu cầu đặt ra trong công đoạn trước không (chúng ta có đang xây dựng đúng đắn sản phẩm không?).

Thanh kiểm tra phần mềm thường là hoạt động kỹ thuật vì nó dùng các kiến thức về các artifacts, các yêu cầu, các đặc tả rời rạc của phần mềm.

Các hoạt động Thanh kiểm tra phần mềm bao gồm kiểm thử (testing) và xem lại (reviews).

Kiểm định phần mềm là qui trình đánh giá phần mềm ở cuối chu kỳ phát triển để đảm bảo sự bằng lòng sử dụng của khách hàng (chúng ta có xây dựng phần mềm đúng theo yêu cầu khách hàng?).

Các hoạt động kiểm định được dùng để đánh giá xem các tính chất được hiện thực trong phần mềm có thỏa mãn các yêu cầu khách hàng và có thể theo dõi với các yêu cầu khách hàng không?

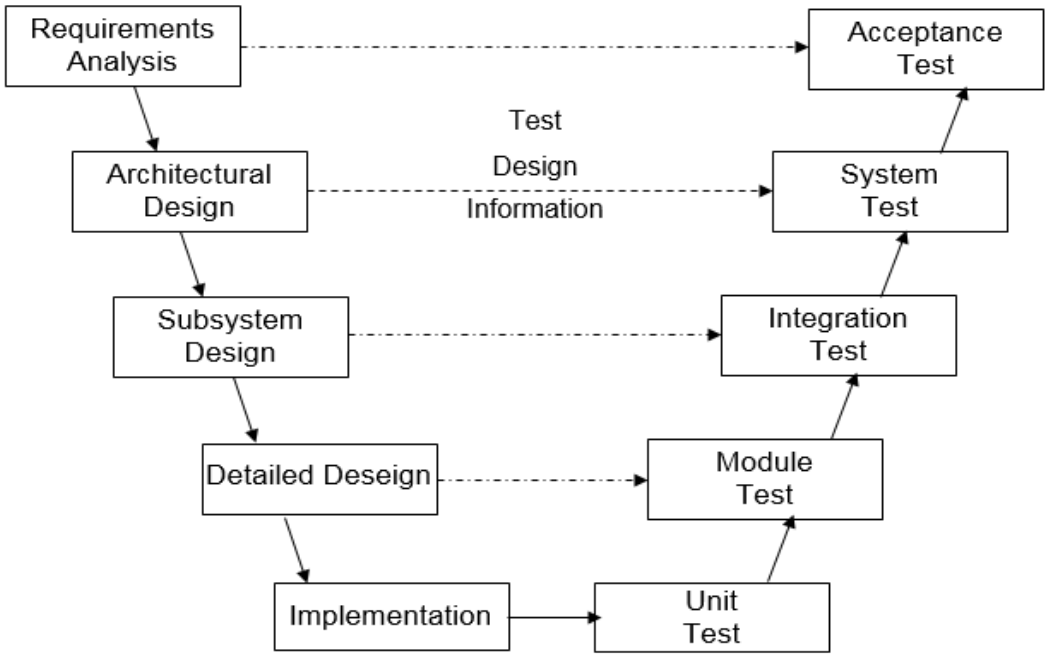
Kiểm định phần mềm thường phụ thuộc vào kiến thức của lĩnh vực mà phần mềm xử lý.

1.1.3. Các mức kiểm thử phần mềm [1]

- Kiểm thử đơn vị (Unit Testing): Kiểm thử đơn vị là việc kiểm thử các đơn vị chương trình một cách độc lập. Thế nào là một đơn vị chương trình? Câu

trả lời phụ thuộc vào ngữ cảnh công việc. Một đơn vị chương trình là một đoạn mã nguồn như hàm hoặc phương pháp của một lớp, có thể được gọi từ ngoài, và cũng có thể gọi đến các đơn vị chương trình khác. Đơn vị cũng còn được coi là một đơn thể để kết hợp. Đơn vị chương trình cần được kiểm thử riêng biệt để phát hiện lỗi trong nội tại và khắc phục trước khi được tích hợp với các đơn vị khác. Kiểm thử đơn vị thường được làm bởi chính tác giả của chương trình, và có thể tiến hành theo hai giai đoạn: kiểm thử đơn vị tĩnh và kiểm thử đơn vị động.

- Kiểm thử module (Module Testing): Kiểm thử các dịch vụ của module có phù hợp với đặc tả của module đó không?



Hình 1.2: Các mức kiểm thử phần mềm

- Kiểm thử tích hợp (Intergration Testing): Mức kế tiếp với kiểm thử đơn vị là kiểm thử tích hợp. Sau khi các đơn vị chương trình để cấu thành hệ thống đã được kiểm thử, chúng cần được kết nối với nhau để tạo thành hệ thống đầy đủ và có thể làm việc. Công việc này không hề đơn giản và có thể có những lỗi về giao diện giữa các đơn vị, và cần phải kiểm thử để phát hiện những lỗi này. Công đoạn này gồm hai giai đoạn: giai đoạn kiểm thử tích hợp và giai đoạn kiểm thử hệ thống. Kiểm thử tích hợp nhằm đảm bảo hệ thống làm việc ổn định trong môi trường thí nghiệm để sẵn sàng cho việc đưa vào môi trường thực sự bằng cách đặt các đơn vị với nhau theo phương pháp tăng dần.

- Kiểm thử hệ thống (System Testing): Kiểm thử mức này được áp dụng khi đã có một hệ thống đầy đủ sau khi tất cả các thành phần đã được tích hợp. Mục đích của kiểm thử hệ thống là để đảm bảo rằng việc cài đặt tuân thủ đầy đủ các yêu cầu được đặc tả của người dùng. Công việc này tốn nhiều công sức, vì

có nhiều khía cạnh về yêu cầu người dùng cần được kiểm thử. Phương pháp kiểm thử hàm là thích hợp nhất cho việc kiểm thử này.

- Kiểm thử chấp nhận (Acceptance Testing): Khi nhóm kiểm thử hệ thống đã thỏa mãn với một sản phẩm, sản phẩm đó đã sẵn sàng để đưa vào sử dụng. Khi đó hệ thống cần phải qua giai đoạn kiểm thử chấp nhận. Kiểm thử chấp nhận được thực thi bởi chính các khách hàng nhằm đảm bảo rằng sản phẩm phần mềm làm việc đúng như họ mong đợi. Có hai loại kiểm thử chấp nhận: kiểm thử chấp nhận người dùng, được tiến hành bởi người dùng, và kiểm thử chấp nhận doanh nghiệp, được tiến hành bởi nhà sản xuất ra sản phẩm phần mềm.

1.1.4. Ca kiểm thử và các phương pháp thiết kế ca kiểm thử

Mỗi ca kiểm thử chứa các thông tin cần thiết để kiểm thử thành phần phần mềm theo một mục tiêu xác định. Thường ca kiểm thử gồm bộ ba thông tin {dữ liệu đầu vào, trạng thái của thành phần phần mềm, dữ liệu đầu ra kỳ vọng} [1].

Dữ liệu đầu vào (Input): Gồm các giá trị dữ liệu cần thiết để thành phần phần mềm dùng và xử lý.

Dữ liệu đầu ra kỳ vọng: Dữ liệu đầu ra mong muốn sau khi thành phần phần mềm xử lý dữ liệu đầu vào.

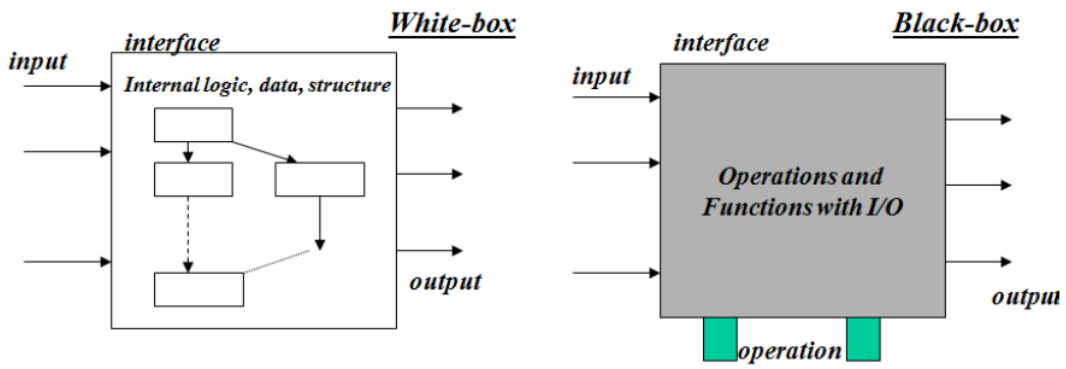
Trạng thái thành phần phần mềm: Được tạo ra bởi các giá trị prefix và postfix.

Tập các ca kiểm thử: Tập hợp các ca kiểm thử mà chúng ta có ý định dùng để kiểm thử thành phần phần mềm để minh chứng rằng thành phần phần mềm có đúng các hành vi mong muốn.

Các phương pháp thiết kế ca kiểm thử

Bất kỳ sản phẩm kỹ thuật nào (phần mềm không phải là ngoại lệ) đều có thể được kiểm thử bởi một trong hai chiến lược [1]:

- Chiến lược kiểm thử hộp đen (Black box Testing): Theo góc nhìn sử dụng
  + Không cần kiến thức về chi tiết thiết kế và hiện thực bên trong.
  + Kiểm thử dựa trên các yêu cầu và đặc tả sử dụng thành phần phần mềm.
- Chiến lược Kiểm thử hộp trắng (White box testing): Theo góc nhìn hiện thực:
  + Cần kiến thức về chi tiết thiết kế và hiện thực bên trong.
  + Kiểm thử dựa vào các lệnh, các nhánh, các điều kiện con, …



Hình 1.3: Minh họa Kiểm thử hộp trắng và hộp đen

Bảng 1.1: Tổng hợp các kỹ thuật kiểm thử hộp đen và hộp trắng được sử dụng ở từng mức kiểm thử

Mức kiểm thử	Chiến lược kiểm thử được dùng
Kiểm thử đơn vị	Hộp trắng, Hộp đen
Kiểm thử tích hợp	Hộp trắng, Hộp đen
Kiểm thử chức năng	Hộp đen
Kiểm thử hệ thống	Hộp đen
Kiểm thử độ chấp nhận của người sử dụng	Hộp đen

1.1.5. Các ý tưởng không đúng về kiểm thử

- Chúng ta có thể kiểm thử phần mềm đầy đủ, nghĩa là đã vét cạn mọi hoạt động kiểm thử cần thiết.
- Chúng ta có thể tìm tất cả lỗi nếu kỹ sư kiểm thử làm tốt công việc của mình.
- Tập các ca kiểm thử tốt phải chứa rất nhiều ca kiểm thử để bao phủ rất nhiều tình huống.
- Ca kiểm thử tốt luôn là ca kiểm thử có độ phức tạp cao.
- Tự động kiểm thử có thể thay thế kỹ sư kiểm thử để kiểm thử phần mềm một cách tốt đẹp.
- Kiểm thử phần mềm thì đơn giản và dễ dàng. Ai cũng có thể làm, không cần phải qua huấn luyện.

1.1.6. Các hạn chế của việc kiểm thử

- Chúng ta không thể chắc chắn là các đặc tả phần mềm đều đúng 100%.
- Chúng ta không thể chắc rằng hệ thống hay tool kiểm thử là đúng.
- Không coi tool kiểm thử nào thích hợp cho mọi phần mềm.
- Kỹ sư kiểm thử không chắc rằng họ hiểu đầy đủ về sản phẩm phần mềm.

- Chúng ta không bao giờ có đủ tài nguyên để thực hiện kiểm thử đầy đủ phần mềm.

- Chúng ta không bao giờ chắc rằng ta đạt đủ 100% hoạt động kiểm thử phần mềm.

1.2. Khái quát về Kiểm thử tĩnh

1.2.1. Định nghĩa về Kiểm thử tĩnh [1]

Việc phân tích tĩnh (kiểm thử tĩnh) được tiến hành dựa trên việc khảo sát các tài liệu được xây dựng trong quá trình phát triển sản phẩm như tài liệu đặc tả nhu cầu người dùng, mô hình phần mềm, hồ sơ thiết kế và mã nguồn phần mềm. Các phương pháp phân tích tĩnh truyền thống bao gồm việc khảo sát đặc tả và mã nguồn cùng các tài liệu thiết kế. Người ta cũng có thể dùng các kỹ thuật phân tích hình thức như kiểm chứng mô hình (model checking) và chứng minh định lý (theory proving) để chứng minh tính đúng đắn của thiết kế và mã nguồn. Công việc này không động đến việc thực thi chương trình mà chỉ duyệt, lý giải về tất cả các hành vi có thể của chương trình khi được thực thi. Tối ưu hóa các chương trình dịch là các ví dụ về phân tích tĩnh.

Định nghĩa kiểm thử tĩnh

Kiểm thử tĩnh là một hình thức của kiểm thử phần mềm mà không chạy chương trình (hoặc phần mềm) được kiểm thử. Điều này ngược với kiểm nghiệm động. Thường thì nó không kiểm thử chi tiết mà chủ yếu kiểm tra tính đúng đắn của code (mã lệnh), thuật toán hay tài liệu.

1.2.2. Phân loại các kỹ thuật kiểm thử tĩnh

Các kỹ thuật kiểm thử tĩnh không tạo ra các ca kiểm thử vì không chạy chương trình được kiểm thử [17]. Các kỹ thuật kiểm thử tĩnh có thể được chia thành hai nhóm kỹ thuật:

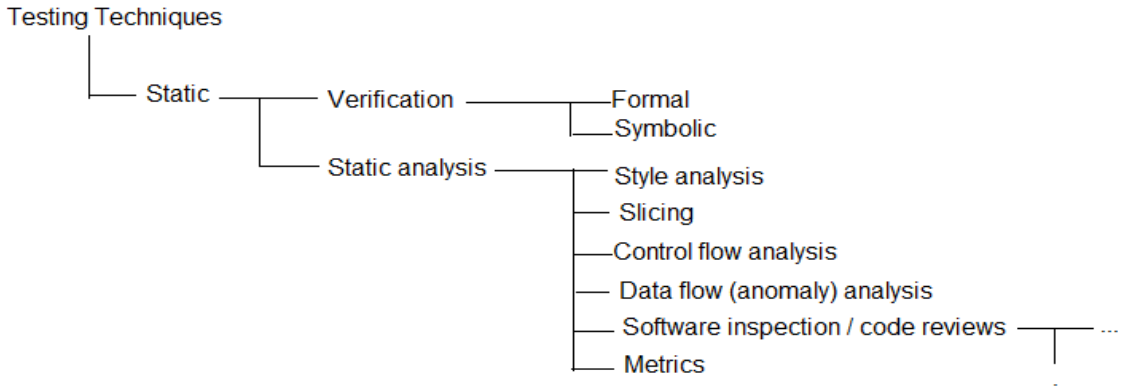
- Nhóm kỹ thuật kiểm thử kiểm tra (verification tests);
- Nhóm kỹ thuật kiểm thử phân tích (analysis tests).

Nhóm kỹ thuật kiểm thử verification có hai kỹ thuật kiểm thử phần mềm: *formally* và *symbolically*. Theo lý thuyết có thể chứng minh một chương trình hành động theo đúng như ý muốn không? Các kỹ thuật kiểm thử tĩnh thông thường được áp dụng để kiểm tra sự chính xác của các module nhỏ trong các chương trình.

Nhóm kỹ thuật phân tích mã tĩnh là một kỹ thuật kiểm tra mã nguồn (source code) hoặc kiểm tra một hình thức mã trung gian nào đó (như bytecode).

Các kỹ thuật kiểm thử phần mềm thuộc nhóm này là *phân tích style (style analysis), slicing, phân tích dòng điều khiển, phân tích dòng (bất thường) dữ liệu* và *code reviews* [17].

Phân loại các kỹ thuật kiểm thử tĩnh được tổng hợp trong Hình 1.4.



Hình 1.4: Phân loại các kỹ thuật kiểm thử tĩnh

Các phần dưới đây, tác giả luận văn sẽ trình bày sơ lược một số kỹ thuật kiểm thử phần mềm thuộc nhóm các kỹ thuật kiểm thử tĩnh.

1.2.3. Sơ lược về các kỹ thuật kiểm thử tĩnh

Các kỹ thuật kiểm thử tĩnh xem xét chương trình tại một thời điểm cụ thể, thông thường không thực thi chương trình. Như trên đã trình bày, các kỹ thuật này được phân thành hai nhóm: *verification* và *phân tích tĩnh (static analysis)*.

Phân tích tĩnh không yêu cầu công cụ hỗ trợ [17]. Code của chương trình được phân tích và được thông dịch [21]. Tuy nhiên, sử dụng công cụ là một đề xuất tốt vì các kỹ thuật phân tích tĩnh phù hợp cho việc tự động hóa.

- ***Phân tích style*** (thông thường cũng được gọi là *kiểm tra từng dòng lệnh của chương trình*) là một kỹ thuật mà kiểm tra mã nguồn xem có thực thi theo đúng mong muốn không. Do tồn tại các plug-in, phân tích style có thể được tích hợp vào một môi trường phát triển tích hợp (IDE). Phân tích style là một phần của các hệ thống nhúng; hơn nữa phân tích tĩnh là một lựa chọn tốt để duy trì vì vậy cũng được gọi là *các hệ thống legacy*.

- ***Phân tích dòng điều khiển (Control flow analysis)*** kiểm tra code để phát hiện các bất thường (anomalies). Mục tiêu của kỹ thuật là phát hiện code mà không được thực thi (*dead code*) và phát hiện các vòng lặp mà không thể thoát khỏi vòng lặp [18].

- ***Phân tích dòng (bất thường) dữ liệu*** **hoặc** ***phát hiện bất thường dòng dữ liệu*** được sử dụng để phát hiện các thành phần của chương trình mà đi lệch với mong muốn. Giống với cách làm việc của một bộ biên dịch, kỹ thuật này cố

gắng phát hiện bất thường trong các thành phần của các chương trình; theo lý thuyết, các thành phần bất thường này cũng có thể được phát hiện bởi những người làm việc cẩn thận [17]. Cụ thể, kỹ thuật này cố gắng kiểm tra các biến đang được định nghĩa, được tham chiếu (tức là được đọc), và chưa được tham chiếu lại (tức là chưa được định nghĩa). Sau khi kiểm tra chương trình, các bất thường như định nghĩa lại các biến mà không đọc chúng… có thể được phát hiện. Bộ biên dịch hỗ trợ loại phân tích này.

- ***Phân tích dòng dữ liệu và dòng điều khiển*** có thể được kết hợp với các công cụ mà duyệt tự động code cho việc phát hiện các bất thường. Chúng có thể được dựa trên vài kỹ thuật khác và có thể cung cấp hỗ trợ giao diện trực quan [22].

1.3. Kết luận

Chương này tác giả luận văn đã trình bày khái quát về kiểm thử phần mềm như: định nghĩa kiểm thử phần mềm, các mức kiểm thử phần mềm... Sau đó tác giả trình bày định nghĩa kiểm thử tĩnh và các kỹ thuật trong kiểm thử tĩnh.

Như vậy chúng ta thấy rằng kiểm thử tĩnh (phân tích tĩnh) là một hình thức của kiểm thử phần mềm. Tuy nhiên, kỹ thuật kiểm thử này không chạy chương trình được kiểm thử. Với phạm vi nghiên cứu của đề tài, trong chương tới tác giả sẽ tập trung trình bày các phương pháp kiểm thử dòng dữ liệu tĩnh thuộc loại kiểm thử tĩnh trong kiểm thử phần mềm.

Chương 2
PHƯƠNG PHÁP KIỂM THỬ DÒNG DỮ LIỆU TĨNH TRONG KIỂM THỬ PHẦN MỀM

Như đã trình bày trong Chương 1, trong chương này tác giả luận văn sẽ tập trung vào nghiên cứu và trình bày các phương pháp kiểm thử dòng dữ liệu tĩnh.

2.1. Phương pháp kiểm thử dòng dữ liệu tĩnh

Chúng ta đã biết, kiểm thử dòng điều khiển và kiểm thử dòng dữ liệu được xem là hai phương pháp chủ yếu trong chiến lược kiểm thử hộp trắng nhằm phát hiện các lỗi tiềm tàng bên trong các chương trình/đơn vị chương trình. Phương pháp kiểm thử dòng điều khiển cho phép sinh ra các ca kiểm thử (tương ứng với các đường đi dòng điều khiển) của chương trình. Tuy nhiên, chỉ áp dụng phương pháp này là chưa đủ để phát hiện tất cả các lỗi tiềm ẩn bên trong chương trình. Trong thực tế, các lỗi thường hay xuất hiện tại các biến được sử dụng trong chương trình/đơn vị chương trình. Kiểm thử dòng dữ liệu cho phép chúng ta phát hiện những lỗi này. Bằng cách áp dụng cả hai phương pháp này, chúng ta khá tự tin về chất lượng của sản phẩm phần mềm [1]. Với phạm vi nghiên cứu của đề tài, trong chương này sẽ trình bày phương pháp kiểm thử dòng dữ liệu tĩnh.

2.1.1. Ý tưởng của phương pháp

Mỗi chương trình/đơn vị chương trình là chuỗi các hoạt động gồm nhận các giá trị đầu vào, thực hiện các tính toán, gán giá trị mới cho các biến (các biến cục bộ và toàn cục) và cuối cùng là trả lại kết quả đầu ra như mong muốn. Khi một biến được khai báo và gán giá trị, nó phải được sử dụng ở đâu đó trong chương trình. Ví dụ, khi khai báo một biến int tem = 0, chúng ta hy vọng biến tem sẽ được sử dụng ở các câu lệnh tiếp theo trong đơn vị chương trình. Việc sử dụng biến này có thể trong các câu lệnh tính toán hoặc trong các biểu thức điều kiện. Nếu biến này không được sử dụng ở các câu lệnh tiếp theo thì việc khai báo biến này là không cần thiết. Hơn nữa, cho dù biến này có được sử dụng thì tính đúng đắn của chương trình chưa chắc đã đảm bảo vì lỗi có thể xảy ra trong quá trình tính toán hoặc trong các biểu thức điều kiện. Để giải quyết vấn đề này, phương pháp kiểm thử dòng dữ liệu xem đơn vị chương trình gồm các đường đi tương ứng với các dòng dữ liệu nơi mà các biến được khai báo, được gán giá trị, được sử dụng để tính toán và trả lại kết quả mong muốn của đơn vị chương trình ứng với đường đi này.

Với mỗi đường đi, chúng ta sẽ sinh một ca kiểm thử để kiểm tra tính đúng đắn của nó. Quá trình kiểm thử dòng dữ liệu được chia thành hai pha riêng biệt: kiểm thử dòng dữ liệu tĩnh (static data flow testing) và kiểm thử dòng dữ liệu động (dynamic data flow testing). Với kiểm thử dòng dữ liệu tĩnh, chúng ta áp dụng các phương pháp phân tích mã nguồn mà không cần chạy chương trình/đơn vị chương trình nhằm phát hiện các vấn đề về khai báo, khởi tạo giá trị cho các biến và sử dụng chúng. Chi tiết về vấn đề này sẽ được trình bày trong phần tiếp theo. Với kiểm thử dòng dữ liệu động, chúng ta sẽ chạy các ca kiểm thử nhằm phát hiện các lỗi tiềm ẩn mà kiểm thử tĩnh không phát hiện được.

2.1.2. Các vấn đề bất thường trong dòng dữ liệu

Trong quá trình lập trình, các lập trình viên có thể viết các câu lệnh "bất thường" hoặc không tuân theo chuẩn lập trình. Chúng ta gọi những bất thường liên quan đến việc khai báo, khởi tạo giá trị cho các biến và sử dụng chúng là các vấn đề về dòng dữ liệu của đơn vị chương trình. Ví dụ, một lập trình viên có thể sử dụng một biến mà không khởi tạo giá trị sau khi khai báo nó *(int x; if (x==100) {...})*.

Các vấn đề bất thường về dòng dữ liệu có thể được phát hiện bằng phương pháp kiểm thử dòng dữ liệu tĩnh. Theo Fosdick và Osterweil [4], các vấn đề này được chia thành ba loại như sau:

- ***Gán giá trị rồi gán tiếp giá trị (Loại 1) [4]:*** Ví dụ hình dưới đây chứa hai câu lệnh tuần tự x = f1(y); x = f2(z); với f1 và f2 là các hàm đã định nghĩa trước và y, z lần lượt là các tham số đầu vào của các hàm này.

```
.
.
.
x = f1(y);
x = f2(z);
.
.
.
```

Hình 2.1: Tuần tự các câu lệnh có vấn đề thuộc loại 1

Chúng ta có thể xem xét hai câu lệnh tuần tự này với các tình huống sau:

- Khi câu lệnh thứ hai được thực hiện, giá trị của biến x được gán và câu lệnh đầu không có ý nghĩa;

- Lập trình viên có thể có nhầm lẫn ở câu lệnh đầu. Câu lệnh này có thể là gán giá trị cho một biến khác như là w = f1(y);

- Có thể có nhầm lẫn ở câu lệnh thứ hai. Lập trình viên định gán giá trị cho một biến khác như là w = f2(z).

- Một hoặc một số câu lệnh giữa hai câu lệnh này bị thiếu. Ví dụ như câu lệnh w = f3(x).

Chỉ có lập trình viên và một số thành viên khác trong dự án mới có thể trả lời một cách chính xác vấn đề trên thuộc trường hợp nào trong bốn tình huống trên. Mặc dù vậy, những vấn đề tương tự như ví dụ này là khá phổ biến và chúng ta cần phân tích mã nguồn để phát hiện ra chúng.

▪ ***Chưa gán giá trị nhưng được sử dụng (Loại 2) [4]:*** Ví dụ hình dưới đây chứa ba câu lệnh tuần tự với y là một biến đã được khai báo và gán giá trị *(y = f(x1)*). Trong trường hợp này, biến z chưa được gán giá trị khởi tạo nhưng đã được sử dụng trong câu lệnh để tính giá trị của biến x (x=y+z).

```
.
.
.
y=f(x1);
int z;
x=y+z;
.
.
.
```

Hình 2.2: Tuần tự các câu lệnh có vấn đề thuộc loại 2

Chúng ta cũng có thể lý giải vấn đề này theo các tình huống sau:

- Lập trình viên có thể bỏ quên lệnh gán giá trị cho biến z trước câu lệnh tính toán giá trị cho biến x. Ví dụ, z = f2(x2), với f2 là một hàm đã được xác định và x2 là một biến đã được khai báo và gán giá trị.

- Có thể có sự nhầm lẫn giữa biến z với một biến đã được khai báo và gán giá trị. Ví dụ: x = y + x2.

▪ ***Đã được khai báo và gán giá trị nhưng không được sử dụng (Loại 3) [4]:*** Nếu một biến đã được khai báo và gán giá trị nhưng không hề được sử dụng (trong các câu lệnh tính toán hoặc trong các biểu thức điều kiện), chúng ta cần xem xét cẩn thận vấn đề này. Tương tự như các trường hợp trên, các tình huống sau có thể được sử dụng để lý giải cho vấn đề này:

- Có sự nhầm lẫn giữa biến này và một số biến khác được sử dụng trong chương trình. Trong thiết kế, biến này được sử dụng nhưng nó đã bị thay thế (do nhầm lẫn) bởi một biến khác.

- Biến này thực sự không được sử dụng trong chương trình. Lúc đầu lập trình viên định sử dụng nó như là một biến tạm thời hoặc biến trung gian nhưng

sau đó lại không cần dùng. Lập trình viên này đã quên xóa các câu lệnh khai báo và gán giá trị cho biến này.

Huang [5] đã giới thiệu một phương pháp để xác định những bất thường trong việc sử dụng các biến dữ liệu bằng cách sử dụng sơ đồ chuyển trạng thái ứng với mỗi biến dữ liệu của chương trình. Các thành phần của sơ đồ chuyển trạng thái của một chương trình ứng với mỗi biến gồm:

- Các trạng thái, gồm:

- U: biến chưa được gán giá trị

- D: biến đã được gán giá trị nhưng chưa được sử dụng

- R: biến đã được sử dụng

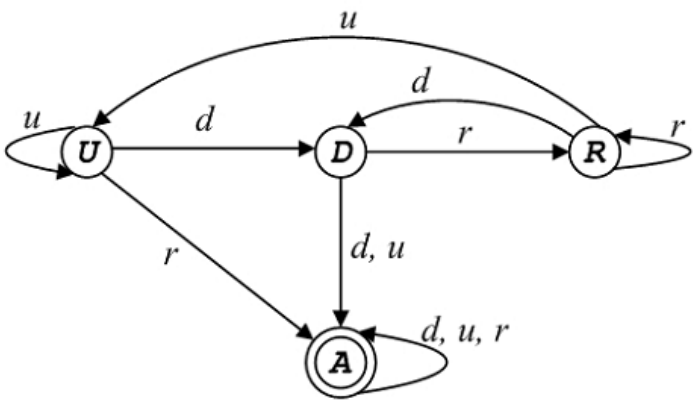
- A: trạng thái lỗi

- Các hành động, gồm:

- d: biến được gán giá trị

- r: biến được sử dụng

- u: biến chưa được gán giá trị hoặc được khai báo lại và chưa được gán giá trị.



Hình 2.3: Sơ đồ chuyển trạng thái của một biến

Hình 2.3 mô tả sơ đồ chuyển trạng thái của một biến trong một chương trình/đơn vị chương trình. Ban đầu, biến này đã được khai báo và chưa được gán giá trị nên trạng thái của chương trình là U. Tại trạng thái này, nếu biến này được sử dụng (hành động r) thì chương trình có vấn đề và trạng thái của chương trình là A. Ngược lại, trạng thái U vẫn được giữ nguyên nếu các câu lệnh tiếp theo vẫn chưa chứa lệnh gán giá trị cho biến này (hành động u). Cho đến khi gặp câu lệnh gán giá trị cho biến này (hành động d), trạng thái của chương trình được chuyển thành D. Nếu biến này được sử dụng ở các câu lệnh tiếp theo (hành

động r) thì trạng thái của chương trình chuyển thành R. Ngược lại, nếu các câu lệnh tiếp theo lại gán lại giá trị cho biến (hành động d) hoặc khai báo lại biến này và không gán giá trị cho nó (hành động u) thì xảy ra vấn đề và trạng thái của chương trình là A. Tại trạng thái này, mọi hành động (d, u và r) xảy ra đều không thay đổi trạng thái của chương trình. Tại trạng thái R, nếu biến này vẫn tiếp tục được sử dụng ở các lệnh tiếp theo (hành động r) thì trạng thái của chương trình vẫn không thay đổi. Ngược lại, nếu xuất hiện câu lệnh gán lại giá trị cho biến (hành động d) thì trạng thái của chương trình quay về D. Trong trường hợp xuất hiện câu lệnh khai báo lại biến này và không gán giá trị cho nó (hành động u) thì chương trình được chuyển từ trạng thái R sang trạng thái U.

Như vậy, các vấn đề với dòng dữ liệu thuộc loại 1 ứng với trường hợp dd xảy ra trong sơ đồ chuyển trạng thái. Các vấn đề thuộc loại 2 ứng với trường hợp ur và loại 3 ứng với trường hợp du. Để phát hiện các vấn đề này, chúng ta sẽ tiến hành xây dựng sơ đồ chuyển trạng thái ứng với mỗi biến như Hình 2.3. Nếu trạng thái A xuất hiện thì chương trình có vấn đề về dòng dữ liệu. Trong trường hợp này, chúng ta cần kiểm tra lại mã nguồn, tìm nguyên nhân của tình huống này và sửa lỗi. Tuy nhiên, cho dù trạng thái lỗi (trạng thái A) không xuất hiện trong quá trình phân tích chương trình, chúng ta vẫn không đảm bảo được rằng chương trình không còn lỗi. Các lỗi có thể xảy ra trong quá trình gán/gán lại giá trị cho các biến và trong quá trình sử dụng chúng (trong các câu lệnh tính toán, trong các biểu thức điều kiện…). Để phát hiện những lỗi này, chúng ta cần phương pháp kiểm thử dòng dữ liệu động.

2.1.3. Phương pháp kiểm thử dòng dữ liệu tĩnh

Một vấn đề lớn của các chương trình được viết ngày nay đó là kiểm soát dữ liệu. Phần lớn các mẫu ngôn ngữ lập trình sử dụng khái niệm của các biến: các đoạn bộ nhớ được đánh dấu có thể được gán (và được gán lại) một giá trị cụ thể, ví dụ một số nguyên hoặc ký tự ASCII. Nhiều biến có thể được sử dụng cùng nhau để tính toán các giá trị của các biến khác; và các biến có thể nhận các giá trị của chúng từ các nguồn khác nhau – như đầu vào con người thông qua bàn phím.

Điều này làm tăng độ phức tạp do vậy có thể dẫn đến có các lỗi bên trong các chương trình: các tham chiếu (references) có thể được tạo ra cho các biến mà không tồn tại các biến, hoặc giá trị của các biến có thể được thay đổi theo khía cạnh nhận giá trị không mong muốn. Khái niệm của kiểm thử dòng dữ liệu tĩnh cho phép người kiểm thử xác định các biến chạy qua chương trình,

giúp người kiểm thử đảm bảo chắc chắn rằng không xảy ra một lỗi mà được miêu tả ở trên.

Kiểm thử dòng dữ liệu tĩnh có thể được xem xét như là một hình thức của kiểm thử cấu trúc (structual testing): ngược với kiểm thử chức năng (kiểm thử hộp đen). Các kỹ thuật kiểm thử cấu trúc yêu cầu người kiểm thử phải truy cập các chi tiết cấu trúc của chương trình. Các biến được định nghĩa và được sử dụng tại các điểm khác nhau bên trong một chương trình; kiểm thử dòng dữ liệu tĩnh cho phép người kiểm thử vẽ đồ thị thay đổi các giá trị của các biến bên trong một chương trình. Kiểm thử dòng dữ liệu tĩnh thực hiện điều này bằng cách sử dụng khái niệm đồ thị chương trình: theo khía cạnh này, kiểm thử dòng dữ liệu tĩnh liên quan đến kiểm thử đường dẫn, tuy nhiên các đường dẫn ở đây được lựa chọn trên các biến.

Có hai hình thức chính của kiểm thử dòng dữ liệu tĩnh: đầu tiên, được gọi là kiểm thử định nghĩa/sử dụng (define/use), sử dụng số lượng các luật đơn giản và các độ đo kiểm thử (test coverage metrics); thứ hai sử dụng "program slices" – các đoạn của một chương trình.

Sự quan trọng của việc phân tích sử dụng các biến trong các chương trình đã được nhận ra từ lâu. Các bộ biên dịch (compilers) cho các ngôn ngữ như COBOL đã giới thiệu tính năng phân tích sử dụng các biến trong chương trình.

Các biến đã được xem như là các vùng chính nơi mà một chương trình có thể được kiểm thử một cách cấu trúc. Các phương pháp trước kia của kiểm thử dữ liệu tĩnh: bộ biên dịch sinh ra một danh sách các dòng tại những biến nào được định nghĩa hoặc được sử dụng. Từ phân tích tĩnh tham chiếu tới thực tế rằng người kiểm thử không chạy chương trình để phân tích chương trình. Phân tích tĩnh cho phép người kiểm thử, theo Jorgensen, tập trung vào ba "bất thường định nghĩa/tham chiếu" đã được trình bày cụ thể ở trên [20].

Ngày nay, phân tích tĩnh vẫn được sử dụng. Cụ thể, tìm kiếm World Wide Web sử dụng các công cụ phân tích luồng dữ liệu tĩnh. Một công cụ, ASSENT, bởi các dịch vụ Tata Consulting Services, được miêu tả như "công cụ phân tích tĩnh luồng dữ liệu toàn cầu cái mà đảm bảo phân tích một cách tự động cho mã Java và C/C++"[1]. Ví dụ khác của công cụ phân tích tĩnh được gọi là LDRA Testbed[2].

[1] Website: http:// www.tcs.com / 0_products /assent / index.htm; description quoted from http: // www.testingfaqs.org / t-static.html#ASSENT

[2] Website: http: // www.ldra.co.uk/

Một phương pháp thay thế đó là xem chương trình theo các biến của chương trình; tuy nhiên, điều này sẽ cắt (slice) chương trình thành các chương trình nhỏ (thành phần) riêng biệt, mỗi thành phần tập trung vào một biến cụ thể tại một vị trí cụ thể bên trong chương trình cần kiểm thử. Khi đó chúng ta có khả năng xác định chương trình theo các biến của chương trình ngoài việc phải xác định toàn bộ chương trình. Khi đó các mối quan hệ và các linkage giữa các biến có thể được xác định một cách dễ dàng hơn, và các slice có thể được kiểm thử một cách độc lập.

Để trình bày phương pháp kiểm thử dòng dữ liệu tĩnh được rõ ràng, luận văn sẽ minh họa thông qua sử dụng ví dụ. Ví dụ này sẽ được tóm lược như sau:

- **Chương trình giảm giá của cửa hàng bán lẻ (staff discount program)**

Chương trình dưới đây được sử dụng trong trường hợp bán lẻ. Chủ cửa hàng đã quyết định rằng hàng hóa của anh ấy/cô ấy giảm 10% trên tất cả các mặt hàng. Nếu các khách hàng mua hơn £15, khi đó tổng giảm giá sẽ tăng 50 xu. Giá của mỗi sản phẩm đang được bán là đầu vào cho chương trình. Khi -1 được nhập vào, tổng giá được hiển thị, cũng như tổng giảm giá và giá cuối cùng được hiển thị.

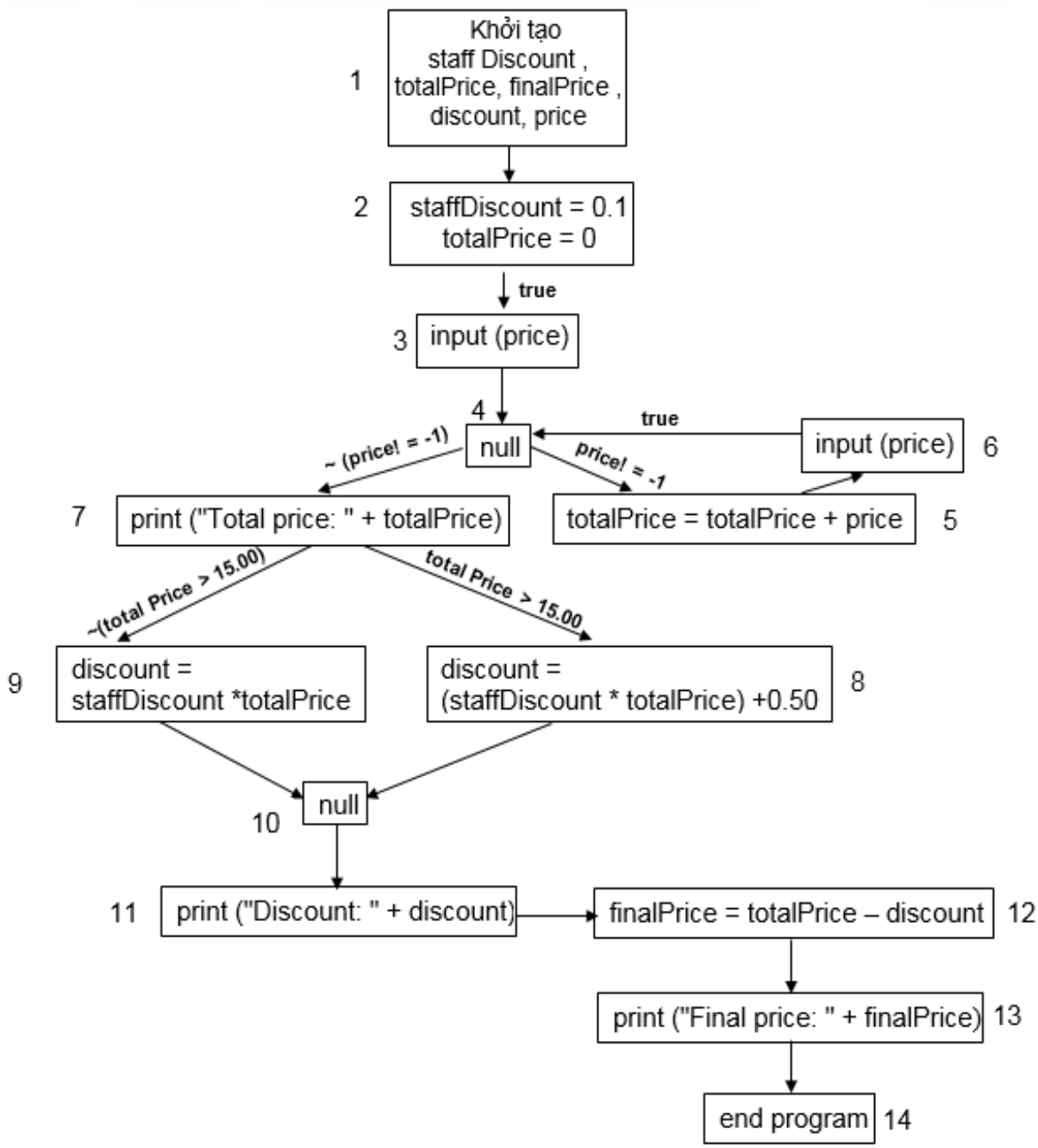
Ví dụ các giá trị £5.50, £2.00 và £2.50 là đầu vào, tổng £10.00. Tổng giảm giá sẽ bằng £1.00 (10% của £10.00), với tổng giá để trả bằng £9.00.

Ví dụ thứ hai sẽ mua £10.50 và £5.00, tổng £15.50. Trong trường hợp này, tổng giá trị trên £15, giảm giá sẽ là £2.05 (10% của £15.50 là £1.55 cộng với 50 xu như vậy tổng gốc trên £15), nghĩa rằng tổng giá phải thanh toán sẽ là £13.45.

Mã nguồn được viết ở dạng giả mã, với một chương trình được viết để thực hiện nhiệm vụ được miêu tả ở trên, được trình bày như sau:

```
program Example()
var staffDiscount , totalPrice, finalPrice , discount, price
staffDiscount = 0.1
totalPrice = 0
input (price)
while (price! = -1) do
        totalPrice = totalPrice + price
        input (price)
od
print ("Total price: " + totalPrice)
if (totalPrice > 15.00) then
discount = (staffDiscount * totalPrice) + 0.50
```

```
else
      discount = staffDiscount * totalPrice
fi
print ("Discount: " + discount)
finalPrice = totalPrice – discount
print ("Final price: " + finalPrice)
end program
```



Hình 2.4: Đồ thị dòng dữ liệu cho chương trình Example

Đồ thị dòng dữ liệu của chương trình trên được trình bày trong Hình 2.4. Mỗi nút trong đồ thị tương ứng với một lệnh định nghĩa, khai báo hoặc sử dụng

biến trong chương trình[3]; tuy nhiên, dòng 1 (trong chương trình) không tương ứng với một nút nào. Điều này là vì các dòng này không sử dụng trong mã thực tế của chương trình: chúng được sử dụng bởi bộ biên dịch để chỉ ra bắt đầu của chương trình và để gán không gian trong bộ nhớ cho các biến tương ứng. Tương tự, các dòng gồm toàn bộ các comment sẽ không được nằm trong đồ thị chương trình.

Các khối mã while và if-then-else trong chương trình được trình bày trên các đồ thị một cách rõ ràng. Các nút 4 tới 6 tương ứng với lặp while, và các nút 7 tới 10 tương ứng với khối if-then-else. Lặp của while được trình bày bởi một lặp từ 4 tới 6; khi biến price bằng -1, dòng dữ liệu đi từ nút 4 đến nút 7.

Đồ thị chương trình thể hiện rằng không có lặp bên trong khối if-then-else: dòng dữ liệu có thể đi từ nút 7 tới nút 8 hoặc nút 9. Điều này tương ứng với các đường dẫn khác nhau mà có thể được theo sau bên trong khối if-then-else: Điều kiện đánh giá là true, tại nút 8 được thi hành, hoặc điều kiện đánh giá là false, nút 9 được thi hành.

Các đồ thị chương trình cho phép người kiểm thử thể hiện cấu trúc của chương trình một cách trực quan. Cấu trúc của các chương trình với nhiều lệnh dòng dữ liệu có thể khó lý giải khi được thể hiện trong hình thức mã nguồn. Sinh ra các đồ thị chương trình cho phép người kiểm thử sử dụng các kỹ thuật kiểm thử dòng dữ liệu tĩnh. Các kỹ thuật kiểm thử dòng dữ liệu tĩnh sẽ được trình bày trong đoạn tiếp theo.

- **Kiểm thử định nghĩa/sử dụng (Define/Use Testing)**

Kiểm thử Define/Use sử dụng các đường dẫn của đồ thị chương trình, liên kết các nút của đồ thị mà liên quan tới các biến, để sinh các ca kiểm thử. Từ "Define/Use" tham chiếu tới hai khía cạnh chính của một biến: biến được định nghĩa (một giá trị được gán tới biến) hoặc biến được sử dụng (giá trị được gán tới biến được sử dụng ngược lại ở chỗ nào đó có thể định nghĩa biến khác). Kiểm thử Define/Use đầu tiên được hình thức hóa bởi Sandra Rapps và Elaine Weyuker vào đầu những năm 1980 [18].

Đồ thị dòng dữ liệu của một chương trình/đơn vị chương trình sử dụng các khái niệm liên quan đến việc định nghĩa và sử dụng các biến.

Định nghĩa của một biến (Def): Một câu lệnh thực hiện việc gán giá trị cho một biến được gọi là câu lệnh định nghĩa biến đó.

---

[3] Chính xác, mỗi nút tương ứng với một đoạn lệnh – cụ thể, trong ngôn ngữ như C, các hành động nhân được kết hợp thành một lệnh, sử dụng một semi-colon như một dấu phẩy. Ví dụ: function1(); function2(); c = function3();. Hơn nữa, các điều kiện nhân có thể được sử dụng trong một lệnh luồng điều khiển. Ví dụ: nếu if (x==2 || y ==3).

Biến được sử dụng (Use): Một câu lệnh sử dụng một biến (để tính toán hoặc để kiểm tra các điều kiện) được gọi là *use* của biến đó.

Với bối cảnh của kiểm thử Define/Use, liên quan đến các biến có hai loại nút: các nút định nghĩa và các nút sử dụng. Các nút được định nghĩa như sau:

Các nút định nghĩa, tham chiếu là DEF(v,n): Nút n trong đồ thị chương trình P là một nút định nghĩa của một biến v trong tập V nếu và chỉ nếu tại n, v được định nghĩa. Ví dụ với biến x, các nút chứa các lệnh như "input x" và "x=2" sẽ là các nút định nghĩa.

Các nút sử dụng, tham chiếu là USE(v,n): Nút n trong đồ thị chương trình P là nút sử dụng của một biến v trong tập V nếu và chỉ nếu tại n, v được sử dụng. Ví dụ với biến x, các nút chứa các lệnh như "print x" và "a=2+x" sẽ là các nút sử dụng.

Các nút sử dụng có thể đươc phân thành các loại nút, phụ thuộc vào cách biến được sử dụng. Cụ thể một biến có thể được sử dụng khi gán một giá trị tới biến khác, hoặc nó có thể được sử dụng khi tạo ra một quyết định mà sẽ ảnh hưởng luồng điều khiển của chương trình.

Hai loại chính của nút sử dụng:

- **P-use:** Một câu lệnh sử dụng biến trong các biểu thức điều kiện (rẽ nhánh, lặp…) được gọi là *p-use* với biến đó.
- **C-use:** Một câu lệnh sử dụng một biến để tính toán giá trị của biến khác được gọi là *c-use* với biến đó (ví dụ b = 3 + d).

Ngoài ra cũng có ba loại khác nữa của nút sử dụng, và tất cả đều thuộc loại C-use:

- **O-use:** sử dụng làm đầu ra – giá trị của biến là đầu ra cho môi trường bên ngoài (ví dụ như màn hình hoặc máy in).
- **L-use:** sử dụng xác định vị trí – giá trị của biến được sử dụng để xác định vị trí nào của một mảng được sử dụng (tức là a[b]).
- **I-use:** sử dụng để lặp – giá trị của biến được sử dụng để điều khiển số lượng các lặp được tạo ra bởi vòng lặp (ví dụ: for (int i = 0; i <= 10; i++)).

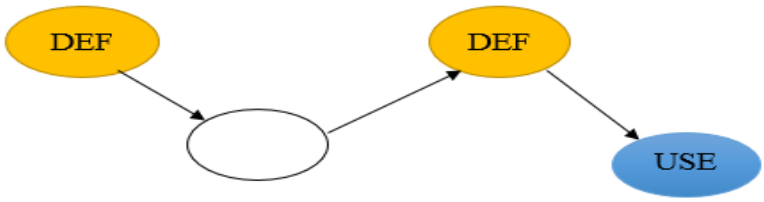
Nhìn vào ví dụ được trình bày ở trên (chương trình giảm giá của cửa hàng), Bảng 2.1 liệt kê nút sử dụng và các nút định nghĩa với biến totalPrice.

Bảng 2.1: Nút sử dụng và nút định nghĩa cho biến totalPrice

Node	Type	Code
2	DEF	**totalPrice** = 0
5	DEF	**totalPrice** = totalPrice + price
5	C-USE	totalPrice = **totalPrice** + price
7	C-USE	print ("Total price: " + **totalPrice**)
(7, 8)	P-USE	(**totalPrice** > 15.00)
(7, 9)	P-USE	~ (**totalPrice** > 15.00)
8	C-USE	discount = (staffDiscount * **totalPrice**) + 0.50
9	C-USE	discount = staffDiscount * **totalPrice**
12	C-USE	finalPrice = **totalPrice** – discount

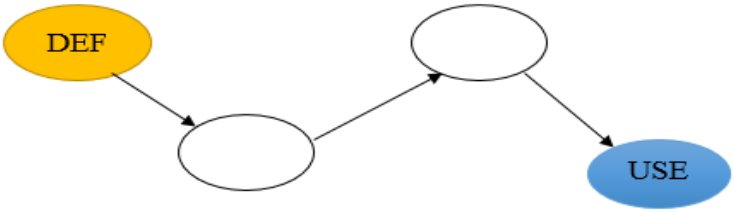
Với các *def* và *use* của biến này, các đường dẫn có ích có thể được sinh ra.

Các đường dẫn Definition-use (du paths): Một đường dẫn trong tập tất cả các đường dẫn trong P(G) là một đường-du cho biến v (trong tập V của tất cả các biến trong chương trình) nào đó nếu và chỉ nếu tồn tại nút DEF(v, m) và nút USE(v, n) thỏa mãn m là nút đầu tiên của đường dẫn - du, và n là nút cuối cùng của đường dẫn – du.



Hình 2.5: Ví dụ của đường DU (DU-path)

Các đường dẫn Definition-clear (dc paths): Một đường dẫn trong tập tất cả các đường dẫn trong P(G) là một đường dẫn - dc cho biến v nào đó (trong tập V của tất cả các biến trong chương trình) nếu và chỉ nếu nó là đường dẫn - du và nút khởi tạo của đường dẫn - dc chỉ là nút định nghĩa của v trong đồ thị chương trình.



Hình 2.6: Ví dụ của đường dẫn – du mà cũng là đường dẫn – dc.

Hình 2.5 trình bày ví dụ của một đường dẫn – du cho biến x trong chương trình. Tuy nhiên, đường dẫn này không là definition-clear vì có một nút định nghĩa thứ hai trong đường dẫn. Hình 2.6 là ví dụ của một đường dẫn – dc.

Nhìn lại ví dụ ở trên, với biến price có các nút định nghĩa và nút sử dụng, được liệt kê dưới Bảng 2.2:

Bảng 2.2: Nút sử dụng và nút định nghĩa cho biến price

Node	Type	Code
3	DEF	input (**price**)
6	DEF	input (**price**)
(4,5)	P-USE	**price**! = -1
(4,7)	P-USE	~ (**price**! = -1)
5	C-USE	totalPrice = totalPrice + **price**

Do vậy, những du-path của biến *price* là: (3 - 4 - 5); (3 - 4 - 7); (6 - 4 - 5); (6 - 4 - 7).

Tất cả các đường dẫn này là definition-clear, do vậy chúng tất cả đều là đường dẫn – dc.

Một phương pháp đơn giản cho việc sinh các đường dẫn – du đó là sử dụng thủ tục Cartesian của tập các nút định nghĩa với tập các nút sử dụng.

- **Các độ đo Rapps-Weyuker**

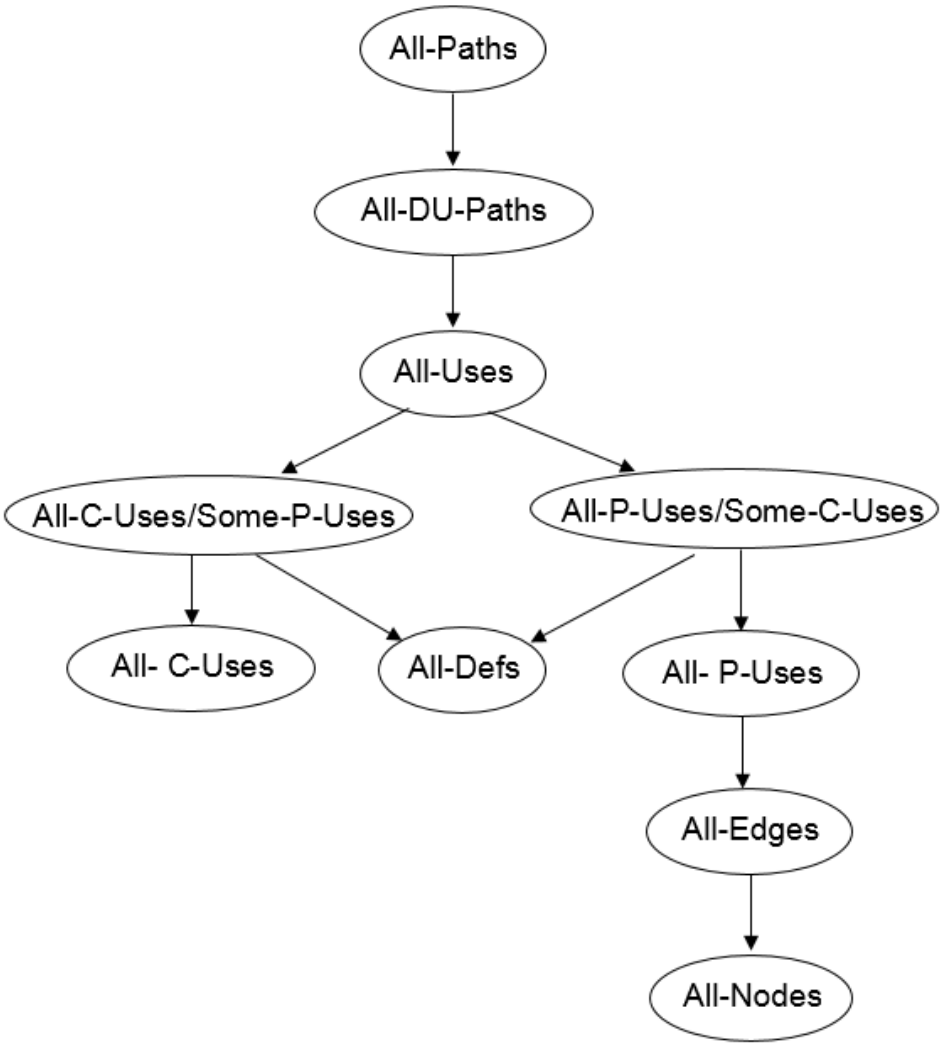
Liên quan đến khái niệm được thảo luận ở các đoạn trên đó là tập các độ đo, cũng được định nghĩa bởi Sandra Rapps và Elaine Weyuker vào đầu những năm 1980 [18]. Các độ đo – tập các tiêu chuẩn, về cơ bản – cho phép người kiểm thử lựa chọn các tập của các đường dẫn của chương trình, ở đây "số lượng các đường dẫn được lựa chọn luôn luôn là giới hạn và được lựa chọn theo cách chuẩn xác và đối xứng để giúp chúng ta loại bỏ các lỗi".

Các đường dẫn của chương trình được lựa chọn, và dữ liệu kiểm thử - là đầu vào cho chương trình - cũng được lựa chọn để bao trùm các đường dẫn này. Có tập các đường dẫn chứa tất cả các đường dẫn có thể có của chương trình (được biết như là tiêu chuẩn All-paths) thường là không thể, vì số lượng các vòng lặp có thể có của chương trình - và do vậy số lượng các đường dẫn có thể có để kiểm thử - thông thường là không giới hạn.

Chín tiêu chuẩn đã được định nghĩa trong các bài báo nghiên cứu. Ba tiêu chuẩn tương ứng cho các độ đo được sử dụng trong kiểm thử đường dẫn, ở đây

các đường dẫn được lựa chọn là không được chọn theo các biến của các đường dẫn và các thuộc tính của các đường dẫn nhưng hơn cả bằng cách phân tích cấu trúc của chương trình. Các độ đo này được biết như là All-Paths (đã được trình bày ở trên), All-Edges và All-Nodes. All-Paths tương ứng với khái niệm 'path coverage', được thỏa mãn nếu mọi đường dẫn của đồ thị chương trình bao hàm trong tập các biến. All-Edges tương ứng với khái niệm 'branch coverage', được thỏa mãn nếu mọi cạnh (branch - nhánh) của đồ thị chương trình được bao hàm. All-Nodes tương ứng với khái niệm 'statement coverage', được thỏa mãn nếu mọi nút được bao hàm bởi tập các đường dẫn. Ngoài ra còn có 6 độ đo mới đã được định nghĩa: All_DU-Paths, All-Uses, All-C-Uses/Some-P-Uses, All-P-Uses/Some-C-Uses, All-Defs và All-P-Uses. Các định nghĩa của các độ đo này được trình bày chi tiết trong [18, 20].

Trong Hình 2.7 các mũi tên thể hiện mối quan hệ giữa các độ đo. Ví dụ All-Paths khỏe hơn All-DU-Paths.



Hình 2.7: Các độ đo Rapps-Weyuker [18]

- **Program Slices**

Khái niệm của program slice đầu tiên được đề xuất bởi Mark Weiser vào đầu những năm 1980 [8, 9]. Theo Weiser, "slice là một sự truyền mã nguồn của một chương trình" [8], cho phép một tập con của một chương trình, tương ứng với một hành vi cụ thể, được lấy ra một cách độc lập. Điều này mang lại lợi ích sau "người lập trình duy trì một chương trình lớn, không quen thuộc" không phải hiểu "toàn bộ hệ thống để thay đổi chỉ một phần nhỏ" [8]. Khái niệm của program slice đã được mở rộng thành duy trì phần mềm bởi Keith Gallagher và James Lyle vào năm 1991 [16], mở rộng các slice trở thành "độc lập số lượng dòng". Các định nghĩa khác của khái niệm program slice được trình bày trong cuốn sách của Paul Jorgensen [20].

Một program slice tương ứng với một biến tại điểm trong chương trình, là tập các lệnh chương trình từ giá trị của biến tại điểm đó của chương trình được tính toán. Định nghĩa này có thể được thay đổi để bao hàm khái niệm đồ thị chương trình: bằng cách thay thế tập các lệnh chương trình với các nút của đồ thị chương trình. Điều này cho phép người kiểm thử tìm danh sách các nút sử dụng từ đồ thị chương trình, và khi đó sinh các slice cùng với các nút sử dụng.

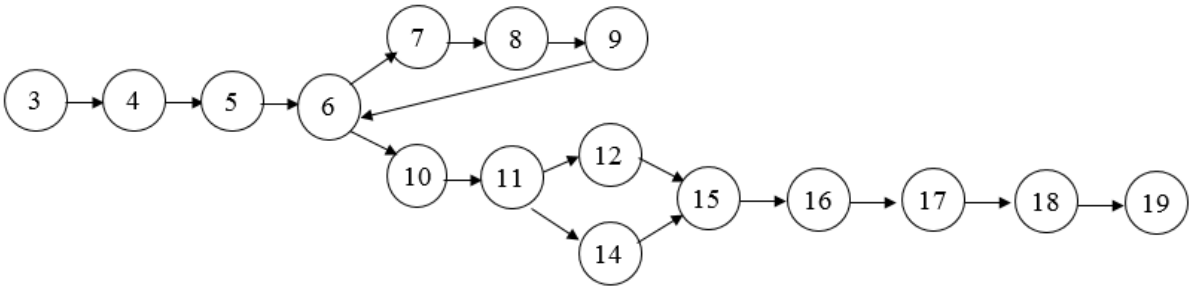
Các program slice sử dụng ký hiệu S(V,n), ở đây S chỉ ra rằng nó là một program slice, V là tập các biến của slice và n tham chiếu tới số lệnh của slice (tức là số nút trong đồ thị chương trình).

Ý tưởng của slice là để tách một chương trình thành các thành phần, mỗi thành phần có một ý nghĩa nhất định.

Áp dụng cho ví dụ về chương trình giảm giá của cửa hàng bán lẻ: trước tiên ta phân mảnh chương trình và xây dựng đồ thị của hàm sau khi phân mảnh (như Hình 2.8):

```
1 program Example()
2 var staffDiscount, totalPrice, finalPrice , discount, price
3 staffDiscount = 0.1
4 totalPrice = 0
5 input (price)
6 while (price! = -1) do
7       totalPrice = totalPrice + price
8       input (price)
9 od
10 print ("Total price: " + totalPrice)
11 if (totalPrice > 15.00) then
```

```
12 discount = (staffDiscount * totalPrice) + 0.50
13 else
14      discount = staffDiscount * totalPrice
15 fi
16 print ("Discount: " + discount)
17 finalPrice = totalPrice – discount
18 print ("Final price: " + finalPrice)
19 end program
```



Hình 2.8: Đồ thị hàm Example sau khi phân mảnh

Các slice cho mỗi việc sử dụng biến price:

- S(price, 5) = {5}
- S(price, 6) = {5, 6, 8, 9}
- S(price, 7) = {5, 6, 8, 9}
- S(price, 8) = {8}

Để sinh slice S(price, 7), các bước dưới đây được thực hiện:

- Dòng 1 đến 4 không bị chịu đựng trên giá trị của biến tại dòng 7, do vậy chúng không được thêm vào slice.

- Dòng 5 chứa một nút định nghĩa của biến price mà có thể ảnh hưởng giá trị tại dòng 7, vì vậy 5 được thêm vào slice.

- Dòng 6 có thể ảnh hưởng giá trị của biến vì nó có thể ảnh hưởng luồng điều khiển của chương trình. Do vậy, 6 được thêm vào slice.

- Dòng 7 không được thêm vào slice, vì nó không thể ảnh hưởng giá trị của biến tại dòng 7.

- Dòng 8 được thêm vào tới slice – thậm chí nó đến sau dòng 7 trong danh sách chương trình. Điều này là vì vòng lặp: sau khi lặp đầu tiên của vòng lặp, dòng 8 sẽ được thực thi trước khi thực thi dòng 7 kế tiếp. Đồ thị chương trình trong Hình 2.8 thể hiện điều này một cách rõ ràng.

- Dòng 9 biểu thị kết thúc của cấu trúc vòng lặp. Điều này ảnh hưởng luồng điều khiển (trình bày trong Hình 2.8, luồng điều khiển đi ngược lại đến

nút 6). Điều này ảnh hưởng không trực tiếp giá trị của price tại dòng 7, do vậy giá trị được lưu trữ trong biến price sẽ được thay đổi tại dòng 8. Do vậy, 9 được thêm vào slice.

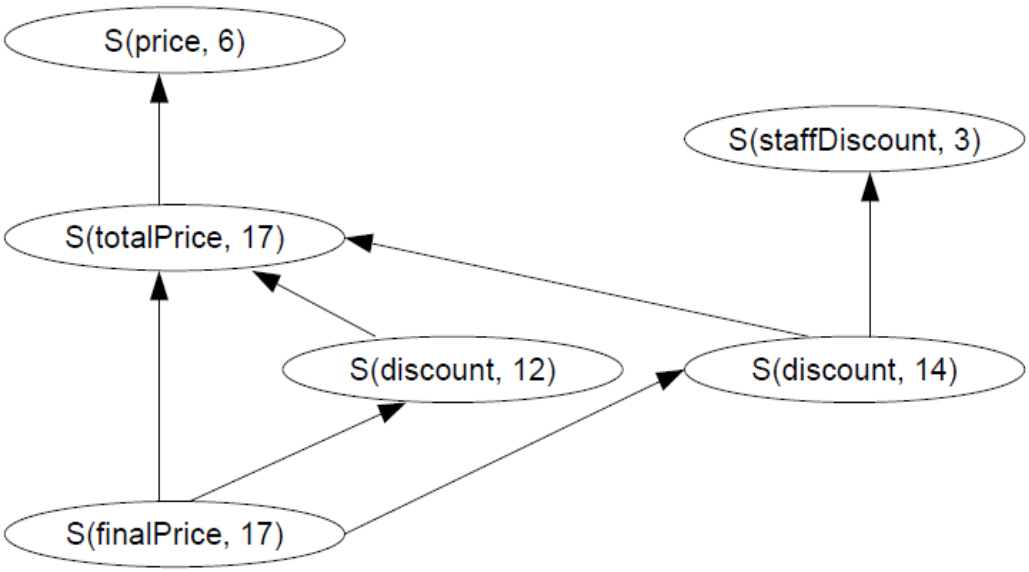
- Không dòng nào của chương trình có thể được thực thi trước dòng 7, và vì vậy không thể ảnh hưởng giá trị của biến tại điểm đó. Do vậy, không dòng nào được thêm vào slice.

Program slice cho phép người lập trình tập trung vào mã cụ thể mà liên quan tới biến cụ thể tại điểm nào đó. Tuy nhiên, khái niệm program slice cũng cho phép người lập trình sinh ra một lưới slice: đó là, một đồ thị thể hiện tập con mối quan hệ giữa các slice khác nhau. Cụ thể, xem ví dụ trước cho biến price, các slice S(price, 5) và S(price, 8) là các tập con của S(price, 7).

Với khía cạnh một chương trình là một chương trình nguyên vẹn, các biến có thể liên quan tới các giá trị của các biến khác: cụ thể, một biến mà chứa một giá trị mà được trả lại tại cuối của thực thi có thể sử dụng các giá trị của các biến khác trong chương trình. Cụ thể, trong ví dụ trên, biến finalPrice sử dụng biến totalPrice, trong đó biến totalPrice sử dụng biến price. Biến finalPrice cũng sử dụng biến discount, trong đó biến discount sử dụng biến staffDiscount và totalPrice… Do vậy, các slice của biến totalPrice và biến discount là tập con của slice của biến finalPrice tại các dòng 17 và 18, do vậy chúng đều phân bố giá trị. Mối quan hệ tập con này ‘ripples down’ tới các biến khác.

Điều này được trình bày trong ví dụ dưới đây:

- S(staffDiscount, 3) = {3}
- S(totalPrice, 4) = {4}
- S(totalPrice, 7) = {4, 5, 6, 7, 8}



Hình 2.9: Program slice lưới

- S(totalPrice, 11) = {4, 5, 6, 7, 8}
- S(discount, 12) = {3, 4, 5, 6, 7, 8, 11, 12}
- S(discount, 14) = {3, 4, 5, 6 ,7, 8, 13, 14}
- S(finalPrice, 17) = {3, 4, 5, 6, 7, 8, 11, 12, 13, 14, 17}

Do vậy, lưới các slice cho biến finalPrice được trình bày trong Hình 2.9. Mối quan hệ này, như đã trình bày trong lược đồ lưới, có thể giúp trong quá trình kiểm thử. Ví dụ nếu có một lỗi trong slice của finalPrice, khi đó bằng cách kiểm thử các slice tập con khác nhau, chúng ta có thể loại trừ chúng từ các nguồn lỗi có thể có. Nếu không có lỗi trong các slice tập con, khi đó lỗi phải được tìm thấy trong các dòng còn lại của code.

2.2. Kết luận

Trong chương này, đầu tiên luận văn trình bày một cách tổng quan về các kỹ thuật kiểm thử dòng dữ liệu tĩnh trong đó đưa ra ý tưởng của kỹ thuật và các vấn đề bất thường trong dòng dữ liệu khi người lập trình code chương trình/đơn vị chương trình.

Như vậy chúng ta thấy rằng kỹ thuật kiểm thử dòng dữ liệu tĩnh được sử dụng ở mức kiểm thử đơn vị và được sử dụng để phát hiện những lỗi trong việc khai báo, sử dụng các biến trong một chương trình/đơn vị chương trình. Nó giúp cho chương trình/đơn vị chương trình chạy ổn định và đưa ra kết quả mong muốn.

Trong [20], tác giả đã chỉ ra một phương pháp phát hiện lỗi trong một chương trình/đơn vị chương trình, đó là sử dụng Logic Hoare trong kiểm thử phần mềm. Chương tới, luận văn sẽ trình bày chi tiết phương pháp phát hiện lỗi này.

Chương 3
ỨNG DỤNG LOGIC HOARE TRONG KIỂM THỬ PHẦN MỀM

Như đã trình bày trong Chương 2, chương này tác giả luận văn tập trung vào trình bày phương pháp ứng dụng Logic Hoare trong kiểm thử phần mềm, cụ thể tác giả trình bày phương pháp kiểm thử kết hợp giữa Logic Hoare với kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu. Đồng thời tác giả cũng áp dụng phương pháp kết hợp này vào một chương trình cụ thể.

3.1. Đặt vấn đề

Chúng ta đã biết hiện nay việc ứng dụng công nghệ thông tin nói chung và phần mềm công nghệ thông tin nói riêng vào trong cuộc sống đã rất rộng và sâu. Dẫn đến vị trí và vai trò của phần mềm rất quan trọng, nó có ảnh hưởng lớn đến sự phát triển của đất nước, tổ chức doanh nghiệp, …. Tuy nhiên, để có một sản phẩm phần mềm tốt và đáp ứng các yêu cầu thì cần phải có nhiều giải pháp và một trong những giải pháp đã và đang được các công ty phát triển phần mềm sử dụng đó là Kiểm thử phần mềm.

Trong Chương 1 và 2, tác giả luận văn đã trình bày khái quát về Kiểm thử phần mềm và các phương pháp kiểm thử phần mềm, đặc biệt đã trình bày chi tiết phương pháp kiểm thử dòng dữ liệu tĩnh. Với phạm vi của đề tài nghiên cứu, trong chương này tác giả sẽ chỉ tập trung trình bày cách ứng dụng Logic Hoare trong kiểm thử phần mềm. Cụ thể là tác giả trình bày phương pháp kiểm thử kết hợp giữa Logic Hoare với kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu.

3.2. Tổng quan về Logic Hoare

Mục tiêu của Logic Hoare là cung cấp một hệ thống hình thức cho việc suy diễn về sự chính xác của chương trình. Logic Hoare dựa trên ý tưởng đặc tả như mối quan hệ giữa hàm (function) và các client của hàm. Đặc tả được tạo nên từ một precondition (tiền điều kiện) và một postcondition (hậu điều kiện). Tiền điều kiện là một vị ngữ miêu tả điều kiện của hàm dựa trên toán tử chính xác; client phải thực thi điều kiện này. Hậu điều kiện là một vị ngữ miêu tả điều kiện hàm thiết lập sau khi chạy một cách chính xác; client có thể dựa trên điều kiện này là đúng sau khi gọi hàm.

Hàm là *chính xác cục bộ (partially correct)* theo đặc tả của hàm nếu, giả thiết tiền điều kiện là true trước khi hàm thực thi, khi đó nếu hàm kết thúc, hậu điều kiện là true. Hàm là *chính xác toàn bộ (totally correct)* nếu, giả thiết tiền điều kiện là true trước khi hàm thực thi, hàm được đảm bảo kết thúc khi hàm thực hiện, hậu điều kiện là true. Do vậy chính xác toàn bộ gồm chính xác cục bộ cộng với kết thúc.

Chú ý rằng nếu một client gọi một hàm mà không thực thi tiền điều kiện của hàm, hàm có thể chạy theo cách nào đó và vẫn chính xác. Do vậy, nếu chúng ta muốn một hàm lỗi nặng, tiền điều kiện có khả năng đầu vào không đúng và hậu điều kiện sẽ miêu tả những gì sẽ xảy ra trong trường hợp đầu vào này.

Logic Hoare sử dụng các bộ ba Hoare (Hoare Triples) để suy diễn về sự chính xác của chương trình. Một bộ ba Hoare có dạng *{P} S {Q}*, ở đây *P* là tiền điều kiện, *Q* là hậu điều kiện, và *S* là các lệnh của hàm. Ý nghĩa của bộ ba *{P} S {Q}* (ở dạng chính xác toàn bộ) là nếu chúng ta bắt đầu ở trạng thái *P* là true và thực thi *S*, khi đó *S* sẽ kết thúc ở trạng thái *Q* là true.

Xét bộ ba Hoare *{x=5} x:=x*2 {x>0}*. Bộ ba này rõ ràng là chính xác, vì nếu *x = 5* và chúng ta nhân *x* với 2 chúng ta có *x = 10*, ẩn ý rõ ràng rằng *x > 0*. Tuy nhiên, mặc dù chính xác nhưng bộ ba Hoare này không chính xác như chúng ta mong muốn. Cụ thể, chúng ta có thể viết hậu điều kiện tốt hơn, tức là ẩn ý rằng *x > 0*. Ví dụ *x > 5 && x < 20* là hậu điều kiện tốt hơn vì nó có nhiều thông tin hơn; nó đưa giá trị *x* chính xác hơn *x > 0*. Postcondtion tốt nhất là *x = 10*; đây là hậu điều kiện có ích nhất. Ở dạng hình thức, nếu *{P} S {Q}* và đối với tất cả *Q* thỏa mãn *{P} S {Q}, Q => Q*, khi đó *Q* là hậu điều kiện tốt nhất của *S* cùng với *P*.

Chúng ta có thể suy diễn theo hướng ngược lại cũng được. Nếu *{P} S {Q}* và với tất cả *P* thỏa mãn *{P} S {Q}, P => P*, khi đó *P* là tiền điều kiện yếu nhất *wp(S, Q)* của *S* cùng với *Q*.

Chúng ta có thể định nghĩa một hàm có tiền điều kiện yếu nhất cùng với hậu điều kiện nào đó cho các phép gán, chuỗi các lệnh, và nếu các lệnh như sau:

wp(x := E, P) = *[E/x]P*
wp(S; T, Q) = *wp(S; wp(T, Q))*
wp(if *B* then *S* else *T, Q*) = *B => wp(S, Q) && ¬B => wp(T, Q)*

Một hàm cũng có thể được định nghĩa cho hậu điều kiện tốt nhất cùng với tiền điều kiện nào đó nhưng điều này yêu cầu các biến có sẵn (để biểu diễn giá trị cũ của biến đang được gán) và dựa vào phạm vi của đề tài nên tác giả sẽ không trình bày vấn đề này.

Để chứng minh chính xác cục bộ của các lặp ở dạng *While B do S*, chúng ta xây dựng một *I* không biến đổi thỏa mãn các điều kiện dưới đây:

P => I: Khởi tạo không biến đổi là true.
{Inv && B} S {Inv}: Mỗi khi thực thi của lặp duy trì không biến đổi.
(Inv && ¬B)=>Q: Không biến đổi và lặp thoát điều kiện postcondition.

Chúng ta có thể chứng minh chính xác toàn bộ bằng cách xây dựng một hàm biến đổi giá trị nguyên (integer) *v* mà đáp ứng các điều kiện dưới đây:

Inv && B => v > 0: Nếu chúng ta vào body của lặp (tức là điều kiện lặp *B* đánh giá là true) và không biến đổi giữ nguyên khi đó *v* phải dương.

{Inv && B && v = V} S {v < V}: Giá trị của hàm biến đổi *v* giảm mỗi lần body của lặp thực thi (ở đây *V* là hằng số).

Chứng minh với Logic Hoare

Xét chương trình WHILE sau:

```
r:= 1
i:= 1
While i < m do
        r:= r*n;
        i:= i + 1
```

Chúng ta muốn chứng minh rằng hàm này tính toán *n* mũ *m* và đưa ra kết quả vào trong *r*. Chúng ta có thể biểu diễn điều này với hậu điều kiện $r = n^m$.

Tiếp theo chúng ta cần xác định tiền điều kiện. Chúng ta không thể tính toán đơn giản nó với *wp* vì chúng ta chưa biết lặp không biến đổi đúng là gì – và thực tế, các lặp không biến đổi khác nhau là có thể dẫn đến các tiền điều kiện khác nhau. Tuy nhiên, một chút suy diễn nhỏ sẽ giúp chúng ta. Chúng ta phải có $m \geq 0$ vì chúng ta không cung cấp cho việc chia bởi *n*, và chúng ta tránh tính toán không chắc chắn 0^0 bằng việc giả thiết $n > 0$. Do vậy tiền điều kiện của chúng ta sẽ là $m \geq 0 \;\&\&\; n > 0$.

Chúng ta cần lựa chọn một lặp không biến đổi (loop invariant). Một heuristic tốt cho việc lựa chọn lặp không biến đổi thường thay đổi hậu điều kiện của lặp không biến đổi để tạo cho lặp không biến đổi phụ thuộc vào chỉ mục lặp (loop index) thay vì sử dụng các biến khác. Do chỉ mục lặp chạy từ *i* tới *m* nên chúng ta có thể thay thế *m* thành *i* trong hậu điều kiện $r = n^m$. Điều này cho chúng ta lặp không biến đổi sẽ gồm có $r = n^i$.

Lặp không biến đổi này sẽ không đủ mạnh (không có đủ thông tin), tuy nhiên vì lặp không biến đổi được kết nối với điều kiện thoát của lặp không biến đổi ẩn ý trong hậu điều kiện. Điều kiện thoát của lặp không biến đổi $i \geq m$ nhưng chúng ta cần biết rằng $i = m$. Chúng ta có thể nhận điều này nếu chúng ta thêm $i \leq m$ tới lặp không biến đổi. Hơn nữa, để chứng minh body của lặp không biến đổi chính xác, chúng ta cần thêm $0 \leq i$ và $n > 0$ tới lặp không biến đổi. Do vậy lặp không biến đổi đầy đủ sẽ là $r = n^i \;\&\&\; 0 \leq i \leq m \;\&\&\; n > 0$.

Để chứng minh chính xác toàn bộ, chúng ta cần biểu diễn một hàm biến đổi (variant function) cho lặp không biến đổi mà hàm biến đổi này có thể được sử dụng để biểu diễn lặp không biến đổi sẽ kết thúc. Trong trường hợp này *m-i* là một lựa chọn tự nhiên vì nó dương và giảm với mỗi lần lặp.

Nhiệm vụ tiếp theo của chúng ta sử dụng các tiền điều kiện yếu nhất để tạo ra nhiệm vụ chứng minh (chứng minh sự chính xác của đặc tả). Đầu tiên chúng ta sẽ đảm bảo rằng lặp không biến đổi ban đầu là true bằng cách truyền lặp không biến đổi qua hai câu lệnh đầu tiên trong chương trình trên:

$\{m \geq 0 \;\&\&\; n > 0\}$
r:= 1;
i:= 0;
$\{r = n^i \;\&\&\; 0 \leq i \leq m \;\&\&\; n > 0\}$

Chúng ta truyền qua lặp không biến đổi *i:= 0* để nhận $r = n^0 \;\&\&\; 0 \leq 0 \leq m \;\&\&\; n > 0$. Chúng ta truyền qua lặp không biến đổi này *r:= 1* để nhận $1 = n^0 \;\&\&\; 0 \leq 0 \leq m \;\&\&\; n > 0$. Do vậy nhiệm vụ chứng minh của chúng ta được biểu diễn như sau:

$m \geq 0 \;\&\&\; n > 0$
$\Rightarrow\; 1 = n^0 \;\&\&\; 0 \leq 0 \leq m \;\&\&\; n > 0$

Chúng ta chứng minh điều này với logic dưới đây:

$m \geq 0 \;\&\&\; n > 0$	bởi giả thiết
$1 = n^0$	vì $n^0 = 1$ với tất cả $n > 0$ và chúng ta có $n > 0$
$0 \leq 0$	bởi định nghĩa của $\leq$
$0 \leq m$	vì $m \geq 0$ bởi giả thiết
n > 0	bởi giả thiết ở trên
$1 = n^0 \;\&\&\; 0 \leq 0 \leq m \;\&\&\; n > 0$	bằng sự kết hợp các giải thích trên

Bây giờ chúng ta áp dụng các tiền điều kiện yếu nhất tới body của lặp không biến đổi. Đầu tiên chúng ta sẽ chứng lặp không biến đổi được duy trì, khi đó chứng minh hàm biến đổi giảm. Để biểu diễn lặp không biến đổi được duy trì, chúng ta có:

$\{r = n^i \;\&\&\; 0 \leq i \leq m \;\&\&\; n > 0 \;\&\&\; i < m\}$
r:= r * n;
i:= i + 1;
$\{r = n^i \;\&\&\; 0 \leq i \leq m \;\&\&\; n > 0\}$

Chúng ta truyền qua lặp không biến đổi i:= i+1 để nhận $r = n^{i+1}$ && $0 \leq i + 1 \leq m$ && $n > 0$. Chúng ta truyền qua lặp không biến đổi r:= r*n để nhận $r * n = n^{i+1}$ && $0 \leq i + 1 \leq m$ && $n > 0$. Do vậy nhiệm vụ chứng minh là:

$r = n^i$ && $0 \leq i \leq m$ && $n > 0$ && $i < m$

⇨ $r * n = n^{i+1}$ && $0 \leq i + 1 \leq m$ && $n > 0$

Chúng ta có thể chứng minh điều này như sau:

$r = n^i$ && $0 \leq i \leq m$ && $n > 0$ && $i < m$	bởi giả thiết
$r * n = n^i * n$	nhân với *n*
$r * n = n^{i+1}$	định nghĩa của mũ
$0 \leq$ i + 1	vì $0 \leq i$
$i + 1 < m + 1$	bởi cộng 1 để xảy ra không bằng
$i + 1 \leq m$	bởi định nghĩa ≤
$n > 0$	bởi giả thiết
$r * n = n^{i+1}$ && $0 \leq i + 1 \leq m$ && $n > 0$	bằng kết nối các giải thích trên

Chúng ta có nhiêm vụ chứng minh để thể hiện rằng hàm biến đổi là dương khi chúng ta thực hiện lặp không biến đổi. Nhiệm vụ chứng minh này là để thể hiện rằng lặp không biến đổi và điều kiện chỉ mục ẩn ý trong:

$r = n^i$ && $0 \leq i \leq m$ && $n < 0$ && $i < m$

⇨ m – i > 0

Chứng minh như sau:

$r = n^i$ && $0 \leq i \leq m$ && $n > 0$ && $i < m$	bởi giả thiết
$i < m$	bởi giả thiết
$m - i > 0$	chuyển *i* về cùng phía

Chúng ta cũng cần thể hiện rằng hàm biến đổi giảm. Chúng ta tạo ra nhiệm vụ chứng minh bằng cách sử dụng các điều kiện tiền điều kiện yếu nhất.

{$r = n^i$ && $0 \leq i \leq m$ && $n > 0$ && $i < m$ && m – i = V}

$r = r * n$

i = i + 1

{m-i < V}

Chúng ta truyền qua điều kiện *i = i + 1* để nhận *m – (i+1) < V*. Truyền qua lệnh kế tiếp không hiệu quả. Do vậy nhiệm vụ chứng minh của chúng ta:

$r = n^i$ && $0 \leq i \leq m$ && $n > 0$ && $i < m$ && $m - i = V$

⇨ $m – (i+1) < V$

Dễ dàng chứng minh:

$r = n^i \;\&\&\; 0 \leq i \leq m \;\&\&\; n > 0 \;\&\&\; i < m \;\&\&\; m - i = V$ bởi giả thiết

$m - i = V$	bởi giả thiết
$m - i - 1 < V$	bởi định nghĩa của <
$m - (i+1) < V$	bởi các luật số học

Cuối cùng chúng ta cần chứng minh rằng hậu điều kiện không thay đổi khi chúng ta thoát khỏi lặp không biến đổi. Chúng ta đã có gợi ý tại sao điều này sẽ là như vậy khi chúng ta chọn lặp không biến đổi. Tuy nhiên, chúng ta cần miêu tả nhiệm vụ chứng minh một cách hình thức:

$r = n^i \;\&\&\; 0 \leq i \leq m \;\&\&\; n > 0 \;\&\&\; i \geq m$

⇨ $r = n^m$

Chúng ta cần chứng minh như sau:

$r = n^i \;\&\&\; 0 \leq i \leq m \;\&\&\; n > 0 \;\&\&\; i \geq m$ bởi giả thiết

$i = m$	vì $i \leq m$ và $i \geq m$
$r = n^m$	thay *i* bằng *m* theo giả thiết

3.3. Ứng dụng Logic Hoare trong kiểm thử phần mềm

Chúng ta biết rằng Logic Hoare được sử dụng để chứng minh sự chính xác của các chương trình trong khi đó kiểm thử là một cách sử dụng trong thực tế để phát hiện các lỗi trong các chương trình. Tuy nhiên việc sử dụng Logic Hoare hiếm khi được áp dụng trong thực tế và kiểm thử cũng khó phát hiện tất cả các lỗi xuất hiện trong các chương trình. Do vậy, trong phần này luận văn sẽ trình bày một kỹ thuật kết hợp Logic Hoare với kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu để nâng cao khả năng phát hiện lỗi cho kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu. Ý tưởng cơ bản của sự kết hợp này như sau: đầu tiên sử dụng kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu để tìm tất cả các đường dẫn chương trình và sau đó sử dụng Logic Hoare để chứng minh sự chính xác của tất cả các đường dẫn chương trình này. Trong quá trình chứng minh, tất cả các lỗi trên các đường dẫn chương trình có thể được phát hiện.

Để thuận tiện cho việc trình bày, dưới dây luận văn sẽ trình bày ở dạng ký hiệu tóm tắt kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu và Logic Hoare để làm nền tảng cho việc trình bày kỹ thuật kết hợp Logic Hoare với kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu.

3.3.1. Sơ lược kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu

Kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu là phương pháp kiểm thử dựa trên đặc tả, sử dụng tiền điều kiện và hậu điều kiện trong việc tạo ra ca kiểm thử [16]. Áp dụng nguyên lý "chia và trị" thì kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu coi đặc tả là *các kịch bản dòng dữ liệu* được tách biệt và tạo ra các tập kiểm thử và phân tích các kết quả kiểm thử dựa trên các kịch bản dòng dữ liệu.

Một kịch bản dòng dữ liệu ở dạng đặc tả kiểu pre-post là một biểu thức logic mà thể hiện một cách rõ ràng rằng điều kiện gì được sử dụng để ràng buộc đầu ra khi đầu vào thỏa mãn điều kiện nào đó.

Cụ thể, cho $S(S_{iv}, S_{ov})[S_{pre}, S_{post}]$ ký hiệu đặc tả của một toán tử S, ở đây S_{iv} là tập tất các biến đầu vào, giá trị của nó không được thay đổi bởi toán tử S, S_{ov} là tập tất cả các biến đầu ra, giá trị của nó được sinh ra hoặc được cập nhật bởi toán tử S, và S_{pre} và S_{post} tương ứng là tiền điều kiện và hậu điều kiện. Đặc điểm của kiểu đặc tả này đó là hậu điều kiện S_{post} được sử dụng để miêu tả mối quan hệ giữa các trạng thái khởi tạo và các trạng thái cuối cùng. Chúng ta giả thiết rằng trong hậu điều kiện một biến như $\tilde{x}$ được sử dụng để biểu thị giá trị khởi tạo của biến x trước khi áp dụng toán tử, tức là x được sử dụng biểu diễn giá trị cuối cùng của x sau khi áp dụng toán tử. Do vậy, $\tilde{x} \in S_{iv}$ và $x \in S_{ov}$. Tất nhiên, S_{iv} cũng chứa tất cả các biến đầu vào được khai báo như các tham số đầu vào và S_{ov} cũng gồm tất cả các biến đầu ra khác được khai báo như các tham số đầu ra.

Một chiến lược thực tế cho tạo ra các ca kiểm thử để thực thi các hành vi mong muốn của tất cả các kịch bản dòng dữ liệu được dẫn từ đặc tả được thiết lập dựa trên khái niệm của kịch bản dòng dữ liệu. Để miêu tả chính xác chiến lược này, đầu tiên chúng ta cần giới thiệu kịch bản dòng dữ liệu.

Định nghĩa 1: Cho $S_{post} \equiv (C_1 \wedge D_1) \vee (C_2 \wedge D_2) \vee \ldots \vee (C_n \wedge D_n)$, ở đây mỗi C_i $(i \in \{1,\ldots,n\})$ là một tiên đề, được gọi là *"guard condition"*, không chứa biến đầu ra trong S_{ov}; D_i là một *"defining condition"* cái mà chứa ít nhất một biến đầu ra trong S_{ov}. Khi đó, một kịch bản dòng dữ liệu f_s của S là một sự kết nối $\tilde{S}_{pre} \wedge C_i \wedge D_i$ và biểu thức $(\tilde{S}_{pre} \wedge C_1 \wedge D_1) \vee)(\tilde{S}_{pre} \wedge C_2 \wedge D_2) \vee \ldots \vee (\tilde{S}_{pre} \wedge C_n \wedge D_n))$ được gọi là kịch bản dòng dữ liệu của S.

Tiền điều kiện $\tilde{S}_{pre} = S_{pre}(\tilde{\sigma}/\sigma)$ ký hiệu kết quả dự đoán từ việc thay thế trạng thái khởi tạo $\tilde{\sigma}$ thành trạng thái cuối cùng σ trong tiền điều kiện S_{pre}. Chúng ta coi sự kết nối $\tilde{S}_{pre} \wedge C_i \wedge D_i$ như là một kịch bản dòng dữ liệu vì nó

định nghĩa hành vi độc lập: khi $\tilde{S}_{pre} \wedge C_i$ được thỏa mãn bởi trạng thái khởi tạo, trạng thái cuối cùng (hoặc các biến đầu ra) được định nghĩa bởi defining condition D_i. Sự kết nối $\tilde{S}_{pre} \wedge C_i$ được biết như là *điều kiện kiểm thử* của kịch bản $\tilde{S}_{pre} \wedge C_i \wedge D_i$, cái này phục vụ như là hệ số cho việc tạo ra ca kiểm thử từ kịch bản dòng dữ liệu này.

Để hỗ trợ việc tạo ra ca kiểm thử một cách tự động từ các kịch bản dòng dữ liệu, bước đầu tiên quan trọng là lấy FSF từ một đặc tả có sẵn. Một thủ tục chuyển có tính hệ thống, thuật toán, và công cụ phần mềm hỗ trợ cho việc dẫn FSF từ đặc tả kiểu pre-post được phát triển trong nghiên cứu [17]. Tạo các ca kiểm thử dựa trên đặc tả sử dụng phương pháp sinh ca kiểm thử dựa trên kịch bản dòng dữ liệu được thực hiện bởi việc tạo ra các ca kiểm thử dựa trên đặc tả từ tất cả các kịch bản dòng dữ liệu của phương pháp sinh ca kiểm thử dựa trên kịch bản dòng dữ liệu. Việc tạo ra các ca kiểm thử từ một kịch bản dòng dữ liệu được thực hiện bằng việc tạo ra các ca kiểm thử từ điều kiện kiểm thử của kịch bản dòng dữ liệu. Trong nghiên cứu [16] tập các tiêu chuẩn cho việc sinh các ca kiểm thử được định nghĩa chi tiết. Để áp dụng KỸ THUẬT DỰA VÀO KỊCH BẢN DÒNG DỮ LIỆU một cách hiệu quả, FSF của đặc tả phải thỏa mãn điều kiện *well-formed* được định nghĩa dưới đây.

Định nghĩa 2: Cho FSF của đặc tả *S* là $(\tilde{S}_{pre} \wedge C_1 \wedge D_1) \vee (\tilde{S}_{pre} \wedge C_2 \wedge D_2) \vee \ldots \vee (\tilde{S}_{pre} \wedge C_n \wedge D_n)$. Nếu *S* thỏa mãn điều kiện $\left(\forall_{i,j \in \{1,\ldots,n\}} \cdot \left(i \neq j \Rightarrow \left(C_i \wedge C_j \Leftrightarrow false\right)\right)\right) \wedge (\tilde{S}_{pre} \Rightarrow (C_1 \vee C_2 \vee \ldots \vee C_n \Leftrightarrow true))$, S được gọi là well-formed.

Well-formed của đặc tả *S* đảm bảo rằng mỗi kịch bản dòng dữ liệu định nghĩa một hàm độc lập và các guard condition bao hoàn toàn lĩnh vực được giới hạn một cách đầy đủ. Do vậy, đối với đầu vào bất kỳ thỏa mãn tiền điều kiện, *S* được đảm bảo để định nghĩa một đầu ra thỏa mãn defining condition của chỉ một kịch bản dòng dữ liệu.

Với giả thiết rằng S là well-formed, chúng ta có thể tập trung vào sinh ca kiểm thử từ một kích bản chức năng đơn, $\tilde{S}_{pre} \wedge C_i \wedge D_i$, tại một thời điểm sử dụng phương pháp của chúng ta. Khi đó ca kiểm thử được sử dụng để chạy chương trình. Chúng ta hãy sử dụng toán tử *ChildFareDiscount.* Chức năng của ChildFareDiscount sử dụng ngôn ngữ đặc tả SOFL [15] dưới đây tương tự như VDM-SL cho các đặc tả toán tử.

Process *ChildFareDiscount(a: int, n_f: int) a_f: int*
Pre *a > 0 and n_f > 1*

Post *(a > 12 => a_f == n_f)*

and

*(a ≤ 12 => a_f == n_f – n_f * 0.5)*

End_process

Đặc tả trên miêu tả rằng đầu vào *a* (đại diện cho *age*) phải lớn hơn *0* và *n_f (normal_fare)* phải lớn hơn 1. Khi *a* lớn hơn 12, đầu ra *a_f (actual_fare)* sẽ bằng *n_f;* ngược lại, *a_f* sẽ giảm bớt 50% trên *n_f.*

Theo thuật toán được báo cáo trong nghiên cứu [4], có ba kịch bản dòng dữ liệu có thể được dẫn từ đặc tả này:

(1) a > 0 and n_f > 1 and a > 12 and a_f = n_f

*(2) a > 0 and n_f > 1 and a ≤ 12 and a_f = n_f – n_f*0.5*

(3) a ≤ 0 or n_f ≤ 1 and anything

Bảng 3.1: Ví dụ kiểm thử

Ca kiểm thử:	*a = 5, n_f = 2*
Điều kiện kiểm thử:	*a > 0 and n_f > 1 and a ≤ 12*
Kịch bản dòng dữ liệu:	*a > 0 and n_f > 1 and a ≤ 12 and a_f = n_f – n_f * 0.5*

Ở đây *anything* nghĩa là bất kỳ điều gì đó có thể xảy ra khi tiền điều kiện bị vi phạm. Giả thiết đặc tả được viết lại theo chương trình dưới đây (giống như Java):

```
    int ChildFareDiscount (int a, int n_f) {
(1) If (a > 0 && n_f > 1) {
(2)    if (a > 12
(3)       a_f:= n_f;
(4)    else a_f:= n_f**2 – n_f – n_f*0.5;
(5) return a_f;}
(6) else System.out.println("precondition bị vi phạm");
(7) }
```

Ở đây ký tự ":=" được sử dụng như là toán tử gán để phân biệt từ ký tự bằng "=" được sử dụng trong đặc tả. Hiển nhiên, chúng ta có thể dẫn các đường dẫn dưới đây: [(1)(2)(3)(5)], [(1)(2)'(4)(5)], và [(1)'(6)]. Trong đường dẫn [(1)(2)'(4)(5)], (2)' nghĩa là thỏa hiệp của điều kiện *a > 12* (tức là *a ≤ 12*), và tương tự cách hiểu áp dụng tới (1)' trong đường dẫn [(1)'(6)]. Chúng ta cũng chèn thêm một số khuyết trong phép gán *a_f = n_f **2 – n_f – n_f * 0.5* (chính xác là *a_f = n_f – n_f * 0.5*), ở đây *n_f **2* nghĩa rằng *n_f* mũ 2 (tức là n_f^2).

Điểm yếu của phương pháp kiểm thử đó là nó chỉ có thể tìm thấy sự có mặt của các lỗi nhưng không tìm thấy sự vắng mặt của các lỗi. Ví dụ, chúng ta sinh ra một ca kiểm thử, *{(a, 5), (n_f, 2)}*, từ điều kiện kiểm thử $a > 0$ *and* $n\_f > 1$ *and* $a \leq 12$ của kịch bản dòng dữ liệu (2), được minh họa trong Bảng 3.1. Thực thi một chương trình với ca kiểm thử này, đường dẫn [(1)(2)'(4)(5)] sẽ được đi qua. Kết quả của thực thi là *a_f = 2**2 – 2 – 2*0.5 = 1*. Kết quả này không chỉ ra sự có sẵn của lỗi vì khi điều kiện kiểm thử $a > 0$ *and* $a\_f > 1$ *and* $a \leq 12$ được thỏa mãn bởi ca kiểm thử, defining condition $a\_f = n\_f - n\_f * 0.5$ cũng được thỏa mãn bởi đầu ra $a\_f = 1$ (vì $1 = 2 - 2*0.5 \Rightarrow true$), cái mà chứng minh mà trong *ca này*, chương trình thực hiện kịch bản dòng dữ liệu một cách chính xác. Nhưng đường dẫn lạ chứa một lỗi.

Một giải pháp cho vấn đề này thực thi một chứng minh dựa trên Logic Hoare để kiểm tra đường dẫn chính xác với kịch bản dòng dữ liệu. Chứng minh chính xác được mong muốn tự động một cách hoàn toàn để cho phép chúng ta tích hợp kỹ thuật này vào trong kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu. Để hiểu hơn, chúng ta cần giới thiệu vắn tắt tiên đề gán trong Logic Hoare.

3.3.2. Ký hiệu được sử dụng trong Logic Hoare

Chúng ta đã biết, Logic Hoare được xây dựng dựa trên logic tiên đề và cung cấp tập các tiên đề để định nghĩa các ngữ nghĩa của các ngôn ngữ lập trình. Với mỗi cấu trúc chương trình, như hệ quả, lựa chọn hoặc lặp một tiên đề cho việc định nghĩa, các ngữ nghĩa của nó được định nghĩa. Các tiên đề này có thể được sử dụng để lý giải về sự chính xác của các chương trình được viết trong một ngôn ngữ lập trình.

Cho *x:= E* là một phép gán: gán kết quả của đánh giá biểu thức *E* tới biến *x*. Tiên đề cho phép gán là:

$$\overline{\{Q(E/x)\}x := E\{Q\}}$$

Biểu thức này thể hiện rằng phép gán *x:=E* là chính xác với post-assertion Q và được dẫn từ pre-assertion $Q(E/x)$, một kết quả dự đoán từ việc thay thế *E* cho tất cả các xảy ra của *x* trong *Q*. Post-assertion *Q* biểu diễn một điều kiện mà phải được thỏa mãn bởi biến *x* sau khi thực thi phép gán (phép gán có thể được coi như là toán tử cập nhật biến *x*). Để tạo ra post-condition Q là true sau khi thực thi, biểu thức *E* phải thỏa mãn *Q* trước khi thực thi, đó là, $Q(E/x)$là true, vì *x* biểu diễn *E* sau khi thực thi.

3.3.3. Kỹ thuật kết hợp Logic Hoare với kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu - Phương pháp TBFV

Kỹ thuật kết hợp Logic Hoare với kỹ thuật kiểm thử dựa trên kịch bản được gọi là *kỹ thuật chứng minh hình thức dựa trên kiểm thử (Testing - Based Formal Verification, viết tắt là: TBFV).* Kỹ thuật TBFV là một kỹ thuật được sử dụng để chứng minh sự chính xác của các đường dẫn chương trình mà những đường dẫn này được xác định bằng kỹ thuật dựa vào kịch bản dòng dữ liệu. Nguyên lý của kỹ thuật TBFV gồm có ba điểm sau:

- Sử dụng kỹ thuật dựa trên kịch bản dòng dữ liệu sinh các ca kiểm thử thích hợp để xác định *tất cả các đường dẫn xuất hiện (representative paths)* trong chương trình được kiểm thử; mỗi đường dẫn được sử dụng ít nhất một ca kiểm thử. Repesentative path được hình thành bằng cách thực hiện một lặp như cấu trúc *if-then-else* để đảm bảo rằng phần thân của lặp được thực hiện ít nhất một lần và lặp kết thúc, và bằng cách thực hiện tất cả các cấu trúc khác giống như hình thức gốc của chúng.
- Cho $\tilde{S}_{pre} \wedge C_i \wedge D_i$ *(i = 1, ..., n)* ký hiệu kịch bản chức năng và ca kiểm thử *t* được sinh ra từ điều kiện kiểm thử $\tilde{S}_{pre} \wedge C_i$. Cho $p = [sc_1, sc_2, \ldots, sc_m]$ là một đường dẫn chương trình, trong đó sc_j *(j = 1,..., m)* được gọi là *đoạn chương trình,* là một quyết định (tức là một dự đoán), một phép gán, một lệnh "return", hoặc một lệnh in. Giả thiết đường dẫn *p* được sử dụng trong ca kiểm thử *t*. Để chứng minh sự chính xác của *p* với kịch bản dòng dữ liệu, chúng ta hình thành một *bộ ba đường dẫn (path triple).*

$$\{\tilde{S}_{pre}\} p \{C_i \wedge D_i\}$$

Bộ ba đường dẫn này giống như như cấu trúc của bộ ba Hoare, nhưng nó được thay đổi thành đường dẫn đơn hơn là chương trình. Điều này có nghĩa rằng nếu tiền điều kiện $\tilde{S}_{pre}$ của chương trình là true trước khi đường dẫn *p* được thực thi, hậu điều kiện $C_i \wedge D_i$ của đường dẫn *p* sẽ là true trên sự kết thúc của *p*.

- Áp dụng lặp đi lặp lại tiên đề phép gán (hoặc tiền đề) chúng ta cung cấp dưới đây cho các câu lệnh liên quan khác, chúng ta có thể dẫn một pre-assertion, được kí hiệu là p_{pre}, để hình thành biểu thức dưới đây:

$$\{\tilde{S}_{pre}(\tilde{x}/x)\}\ \{p_{pre}(\tilde{x}/x)\}\ p\ \{C_i \wedge D_i(\tilde{x}/x)\}$$

Ở đây $\tilde{S}_{pre}(\tilde{x}/x)$, $p_{pre}(\tilde{x}/x)$ và $C_i \wedge D_i(\tilde{x}/x)$ tương ứng là kết quả dự đoán tương ứng từ việc thay thế mọi biến đầu vào $\tilde{x}$ tương ứng cho biến đầu

vào *x* trong dự đoán. Các sự thay thế này là cần thiết để loại bỏ xung đột giữa các biến đầu vào và các biến được cập nhật bên trong.

Cuối cùng, nếu $\tilde{S}_{pre}(\tilde{x}/x) \Rightarrow p_{pre}(\tilde{x}/x)$ được chứng minh có nghĩa rằng không có lỗi nào xuất hiện trên đường dẫn; ngược lại chỉ ra sự xuất hiện lỗi trên đường dẫn.

Tiên đề cho các lệnh liên quan khác hoặc các quyết định liên quan khác được đưa ra như sau:

$$\overline{\{Q\}S\{Q\}}$$

Ở đây *S* là một trong ba loại phân đoạn chương trình: *lệnh quyết định, lệnh "return" và lệnh in.* Tiên đề này miêu tả tiền điều kiện và hậu điều kiện cho một trong ba loại phân đoạn chương trình vì không phân đoạn chương trình nào thay đổi trạng thái. Chúng ta gọi tiên đề này là *tiên đề cho phân đoạn không thay đổi*.

Chúng ta thấy rằng do ứng dụng của các tiên đề phép gán và phân đoạn không thay đổi chỉ gồm có thao tác bằng tay theo cú pháp, dẫn từ pre-assertion $p_{pre}(\tilde{x}/x)$ có thể được thực hiện một cách tự động, nhưng ẩn ý chứng minh một cách hình thức $\tilde{S}_{pre}(\tilde{x}/x) \Rightarrow p_{pre}(\tilde{x}/x)$, chúng ta có thể viết đơn giản như sau $\tilde{S}_{pre} \Rightarrow p_{pre}$ trong báo cáo này, không thể thực hiện một cách tự động, thậm chí với sự hỗ trợ của một bộ chứng minh lý thuyết, phụ thuộc vào độ phức tạp của $\tilde{S}_{pre}$ và p_{pre}. Nếu thu được một cách tự động đầy đủ theo ưu tiên cao nhất, chứng minh hình thức của ẩn ý này có thể được "thay thế" bởi một ca kiểm thử. Đó là, đầu tiên chúng ta sinh ra các giá trị mẫu cho các biến trong $\tilde{S}_{pre}$ và p_{pre}, và khi đó đánh giá chúng xem p_{pre} là false khi $\tilde{S}_{pre}$ là true. Nếu điều này là true, chúng ta nói rằng đường dẫn đang được chứng minh chứa một lỗi. Do kỹ thuật kiểm thử có sẵn trong các bài báo [16, 19] nên không cần trình bày lại chi tiết trong luận văn này.

3.4. Áp dụng phương pháp TBFV

3.4.1. Áp dụng cho đoạn chương trình

Để thử nghiệm phương pháp TBFV, trong đoạn này tác giả luận văn sẽ trình bày một nghiên cứu áp dụng phương pháp TBFV để kiểm thử và chứng minh đoạn chương trình của *IC card system (hệ thống thẻ IC)* cho *JR commute train service (dịch vụ tàu điện chuyển mạch JR)* ở Tokyo. Qua thử nghiệm thấy rằng phương pháp TBFV về cơ bản sử dụng được và hiệu quả nhưng phương pháp này cũng đối diện vài thách thức hoặc vài giới hạn mà cần được giải quyết trong các nghiên cứu tiếp theo.

Hệ thống thẻ IC được thiết kế để cung cấp các dịch vụ chức năng sau: (1) Điều khiển truy cập đến và thoát từ một trạng thái đường ray, (2) Mua vé sử dụng thẻ IC, (3) Nạp thẻ IC bằng tiền mặt hoặc thông qua tài khoản ngân hàng, và (4) Mua vé đi lại trong khoảng thời gian (thời gian một tháng hoặc ba tháng). Do giới hạn về thời gian, tác giả không thể trình bày chi tiết tất cả, nhưng tác giả sẽ lấy một trong các hoạt động bên trong được sử dụng trong hệ thống thẻ IC, gọi là *ChildFareDiscount* đã được trình bày ở trên, làm ví dụ để minh họa cách phương pháp TBFV được áp dụng. Chương trình ChildFareDiscount chứa ba đường dẫn, chúng ta cần chứng minh hình thức ba đường dẫn. Do vậy quá trình chứng minh cho ba đường dẫn này là giống nhau nên tác giả chỉ cần chứng minh đường dẫn [(1)(2)'(4)(5)], đường dẫn này được sử dụng bởi ca kiểm thử *{(a, 5), (n_f, 2)}*.

Đầu tiên chúng ta xây dựng bộ ba đường dẫn:

$\{\tilde{a} > 0 \; and \; \tilde{n\_f} > 1\}$
$[\; a > 0 \;\&\&\; n\_f > 1$
$\; a \leq 12,$
$\; a\_f := n\_f ** 2 - n\_f - n\_f * 0.5,$
$\; return \; a\_f \;]$
$\{\; \tilde{a} \leq 12 \; and \; a\_f = \tilde{n\_f} - \tilde{n\_f} * 0.5$

Ở đây $\tilde{a} > 0$ và $\tilde{n\_f} > 1$ là kết quả thay thế $\tilde{a}$ và $\tilde{n\_f}$ cho các biến đầu vào a và n_f tương ứng trong pre-condition của chương trình, và $\tilde{a} \leq 12$ và $a\_f = \tilde{n\_f} - \tilde{n\_f} * 0.5$ là kết quả hoàn thành thay thế trong post-condition.

Thứ hai, chúng ta áp dụng lặp đi lặp lại tiên đề phép gán hoặc tiền đề phép gán với lệnh không thay đổi cho bộ ba đường dẫn [(1)(2)'(4)(5)], bắt đầu từ post-condition. Kết quả chúng ta xây dựng được đường dẫn dưới đây, được gọi là *asserted path (đường dẫn đã được chứa thêm các khẳng định),* với các khẳng định bên trong được dẫn từ hai đoạn chương trình*:*

$\{\; \tilde{a} > 0 \; and \; \tilde{n\_f} > 1\}$
$\{\; \tilde{a} \leq 12 \; and$
$\tilde{n\_f} ** 2 - \tilde{n\_f} - \tilde{n\_f} * 0.5 = \tilde{n\_f} - \tilde{n\_f} * 0.5\}$
$\; a > 0 \;\&\&\; n\_f > 1$
$\{\; \tilde{a} \leq 12 \; and$
$n\_f ** 2 - n\_f - n\_f * 0.5 = \tilde{n\_f} - \tilde{n\_f} * 0.5\}$

$\; a \leq 12$
$\{\tilde{a} \leq 12 \; and$
$n\_f ** 2 - n\_f - n\_f * 0.5 = \tilde{n\_f} - \tilde{n\_f} * 0.5\}$

$a\_f := n\_f ** 2 - n\_f - n\_f * 0.5$
$\{\tilde{}a \leq 12$ *and* $a\_f = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$
return a_f
$\{\tilde{}a \leq 12$ *and* $a\_f = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$

Ở đây đường dẫn quyết định $\tilde{}a \leq 12$ và $\tilde{}n\_f ** 2 - \tilde{}n\_f - \tilde{}n\_f * 0.5 = \tilde{}n\_f - \tilde{}n\_f * 0.5$, dòng thứ hai từ trên xuống, là kết quả thay thế $\tilde{}a$ cho *a* và $\tilde{}n\_f$ cho *n_f* trong quyết định $\{\tilde{}a \leq 12$ và $n\_f ** 2 - n\_f - n\_f * 0.5 = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$. Như đã trình bày ở trên, điều này là cần thiết để giữa sự tin cậy của các biến đầu vào *a* và *n_f* trong pre-condition gốc (biểu thị là $\tilde{}a$ và $\tilde{}n\_f$) và pre-assertion.

Thứ ba, chúng ta cần đánh giá tính hợp lý của ẩn ý $\tilde{}a > 0$ và $\tilde{}n\_f > 1 \Rightarrow \tilde{}a \leq 12$ và $\tilde{}n\_f ** 2 - \tilde{}n\_f - \tilde{}n\_f * 0.5 = \tilde{}n\_f - \tilde{}n\_f * 0.5$. Sử dụng ca kiểm thử $\{(\tilde{}a, 5), (\tilde{}n\_f, 8)\}$, chúng ta có thể dễ dàng chứng minh rằng ẩn ý này là sai (đánh giá chi tiết không được đề cập vì giới hạn thời gian).

Từ ví dụ trên, chúng ta có thể thấy rằng đôi khi kiểm thử thậm chí có thể hiệu quả hơn chứng minh hình thức trong việc đánh giá tính hợp lý của ẩn ý khi một lỗi có sẵn trên đường dẫn, nhưng nếu đường dẫn không chứa lỗi, về cơ bản kiểm thử sẽ không thể phát hiện để đưa ra một kết luận. Trong trường hợp này, một đánh giá kỹ thuật phải được tạo ra cho việc đánh giá tính hợp lý. Điểm mạnh của kiểm thử đó là có thể thực hiện được tự động, đây là điều vô cùng có ích trong thời đại công nghiệp.

3.4.2. Áp dụng cho việc gọi phương thức

Nếu một gọi phương thức (method invocation) được sử dụng như là câu lệnh, chúng ta có thể thay đổi trạng thái hiện tại của chương trình *ChildFareDiscount*. Do vậy, đường dẫn bên trong phương thức được gọi sẽ phải được xem xét trong pre-assertion của chương trình dưới dạng kiểm thử.

Chúng ta hãy thay đổi chương trình *ChildFareDiscount* và tổ chức hoàn thiện chương trình này thành một lớp (class) được gọi là *FareDiscount* dưới đây.

```
class FareDiscount{
int tem; //instance variable
int ChildFareDiscount1(int a, int n_f){
(1)   Discount(n_f);
(2)   if (a > 0 && n_f > 1){
(3)   if (a > 12)
(4)      a_f := n_f
(5)   else a_f := n_f **2 – n_f – tem;
```

```
(6)     return a_f; }
(7)     else System.out.println("the precondition is violated.");
        }

        void Discount(int x){
        int r;
(1.1)   r := x*0.5;
(1.2)   tem := r; }
        }
```

Khi chạy phương thức *ChildFareDiscount*1 trong đó phương thức *Discount(n_f)* được gọi, chúng ta lấy được ba đường dẫn: [(1)(2)(3)(4)(6)], [(1)(2)(3)'(5)(6)] và [(1)(2)'(7)], ở đây đoạn (1) là một đường dẫn con (subpath) [(1.1)(1.2)](n_f/x), biểu thị đường dẫn kết quả từ việc thay thế tham số thực tế *n_f* cho tham số hình thức *x* trong đường dẫn con [(1.1)(1.2)]. Do vậy đường dẫn [(1)(2)(3)'(5)(6)] thực tế sau khi chèn thêm đường dẫn trong *Discount* vào đường dẫn trong *ChildFareDiscount*1 được biểu diễn như sau [(1.1)(1.2)(2)(3)'(5)(6)]. Lựa chọn ca kiểm thử giống nhau {(a, 5), (n_f, 2)} trước khi chạy chương trình, chúng ta tạo ra đường dẫn [(1.1)(1.2)(2)(3)'(5)(6). Khi đó chúng ta xây dựng được đường dẫn *asserted path* như sau:

$\{\tilde{}a > 0 \ \ and \ \ \tilde{}n\_f > 1\}$
$\{\tilde{}a \leq 12 \ and$
$\tilde{}n\_f **2 - \tilde{}n\_f - \tilde{}n\_f * 0.5 = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$
$r := n\_f * 0.5$
$\{\tilde{}a \leq 12$
$n\_f **2 - n\_f - r = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$
$tem := r$
$\{\tilde{}a \leq 12 \ and$
$n\_f **2 - n\_f - tem = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$
$a > 0 \ \&\& \ n\_f > 1$
$\{\tilde{}a \leq 12 \ and$
$n\_f **2 - n\_f - tem = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$
$a \leq 12$
$\{\tilde{}a \leq 12 \ and$
$n\_f **2 - n\_f - tem = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$
$a\_f := n\_f **2 - n\_f - tem$
$\{\tilde{}a \leq 12 \ and \ a\_f = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$
$return \ \ a\_f$

$\{\tilde{}a \leq 12 \; and \; a\_f = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$

Ở đây đường dẫn con *[r:= n_f *0.5, tem:= r]* là kết quả thay thế tham số thực tế *n_f* được sử dụng trong lời gọi phương thức *Discount(n_f)* cho tham số hình thức *x* được sử dụng trong định nghĩa phương thức trong đường dẫn con gốc *[r:= x*0.5, tem:= r]*. Tương tự, chúng ta có thể dễ dàng sử dụng kiểm thử để chứng minh ẩn ý $\tilde{}a > 0$ và $\tilde{}n\_f > 1 \Rightarrow \tilde{}a \leq 12$ và $\tilde{}n\_f ** 2 - \tilde{}n\_f - \tilde{}n\_f * 0.5 = \tilde{}n\_f - \tilde{}n\_f * 0.5$ là sai, chỉ ra rằng một lỗi được tìm thấy trên đường dẫn.

3.4.3. Các nghiên cứu liên quan

Qua nghiên cứu thấy các nghiên cứu dựa trên việc tích hợp Logic Hoare và kiểm thử dường như tập trung chủ yếu vào sử dụng pre-assertion và post-assertion trong bộ ba Hoare cho việc sinh ca kiểm thử và phân tích kết quả kiểm thử và không có nghiên cứu nào giống như phương pháp TBFV cho việc giải bài toán kiểm thử dựa trên đặc tả.

Một trong những kết quả đạt được là trình bày Design By Contract (DBC) của Meyer đã được ứng dụng trong ngôn ngữ lập trình Eiffel [7, 8]. Thành công của Eiffel đó là kiểm tra pre-condition và post-condition và khuyến khích môn học DBC trong lập trình để phát triển nghiên cứu tương tự cho các ngôn ngữ khác như hệ thống kiểm thử Sunit cho Smalltalk [13]. Cheon và Leavens miêu tả một phương pháp kiểm thử đơn vị (unit testing) mà sử dụng một bộ kiểm tra assertion thời gian chạy của ngôn ngữ đặc tả hình thức để quyết định các phương thức làm việc chính xác theo đặc tả hình thức sử dụng pre-condition và post-condition, và đã cài đặt thành công ý tưởng này sử dụng ngôn ngữ mô hình Java (Java Modeling Language – JML) và nền tảng làm việc kiểm thử Junit [20]. Gray và Mycroft miêu tả phương pháp khác để kiểm thử các chương trình Java sử dụng các đặc tả kiểu Hoare [18]. Họ đã trình bày cách các đặc tả kiểm thử logic với một post-condition được nhúng vào Java và cách ngôn ngữ đặc tả kiểm thử có thể được biên dịch vào trong Java cho việc thực thi chương trình đúng. Ngoài ra còn có nhiều các kết quả tương tự trong các bài báo tuy nhiên do thời gian có hạn nên tác giả luận văn chỉ trình bày các kết quả tiêu biểu.

3.5. Kết luận

Trong chương này, tác giả luận văn đã trình bày một phương pháp chứng minh hình thức dựa trên kiểm thử (TBFV) cho việc phát hiện lỗi trong các chương trình bằng cách tích hợp kiểm thử dựa trên đặc tả và Logic Hoare. Nguyên tắc cơ bản của TBFV đó là trước tiên sử dụng kiểm thử dựa trên kịch

bản dòng dữ liệu để đưa ra đường dẫn của chương trình dưới hình thức kiểm thử, và khi đó áp dụng phương pháp dựa Logic Hoare để chứng minh hình thức sự chính xác của mỗi đường dẫn. Do kỹ thuật cho kỹ thuật dựa trên kịch bản dòng dữ liệu và kỹ thuật cho chứng minh sự chính xác có thể được thực hiện tự động nên phương pháp TBFV có một ưu điểm so với chứng minh sự chính xác hình thức dựa trên Logic Hoare đó là có thể thực hiện tự động bằng cách xây dựng hệ thống chương trình thực tế. Phương pháp này cũng có ưu điểm nổi bật trong việc giảm số lượng các ca kiểm thử cần thiết so với kiểm thử dựa trên đặc tả có sẵn.

Trong khi tập trung vào trình bày ý tưởng cơ bản của phương pháp TBFV và một ví dụ để trình bày tính hiệu quả nổi bật và tiện lợi trong báo cáo này, một thử nghiệm cần được xây dựng để đánh giá hiệu quả có tính hệ thống và để so sánh với các kiểm thử liên quan và các phương pháp chứng minh hình thức. Nghiên cứu tương lai cũng cần được giải quyết đưa ra công cụ hỗ trợ.

KẾT LUẬN VÀ KIẾN NGHỊ

1. Kết luận

Từ việc nghiên cứu Tổng quan về kiểm thử phần mềm và kiểm thử tĩnh để nắm những kiến thức cơ sở về kiểm thử phần mềm nói chung và kiểm thử tĩnh nói riêng phục vụ các nghiên cứu tiếp theo. Sau đó, tác giả luận văn tiến hành nghiên cứu khái quát các phương pháp kiểm thử dòng dữ liệu tĩnh trong kiểm thử phần mềm. Cuối cùng tác giả nghiên cứu Logic Hoare và ứng dụng Logic Hoare trong kiểm thử phần mềm, cụ thể là tác giả trình bày phương pháp kiểm thử kết hợp giữa Logic Hoare với kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu để nâng cao hiệu quả kiểm thử của kỹ thuật dựa trên kịch bản dòng dữ liệu và áp dụng phương pháp kết hợp này vào kiểm thử một chương trình cụ thể.

Như vậy với quá trình nghiên cứu ở trên về mặt cơ bản em đã hoàn thành được mục tiêu của đề tài đưa ra. Một số kết quả đạt được như sau:

- Nắm được kiến thức cơ bản liên quan đến Kiểm thử phần mềm và kiểm thử tĩnh;
- Nắm được các kỹ thuật kiểm thử tĩnh và các phương pháp kiểm thử dòng dữ liệu tĩnh trong kiểm thử phần mềm;
- Hiểu được Logic Hoare trong việc chứng minh sự chính xác của chương trình và nghiên cứu được kỹ thuật kết hợp Logic Hoare với kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu để nâng cao hiệu quả cho kỹ thuật kiểm thử dựa trên kịch bản dòng dữ liệu.
- Báo cáo có thể làm tài liệu tham khảo về lĩnh vực Kiểm thử phần mềm, Kiểm thử tĩnh và đặc biệt là kiểm thử dòng dữ liệu tĩnh;
- Kết quả nghiên cứu có thể làm tiền đề cho các nghiên cứu liên quan khác.

2. Kiến nghị

Với thời gian nghiên cứu ngắn và đây là lĩnh vực mới tiếp cận nên trong báo cáo còn một số phần chưa được hoàn thiện. Do vậy thời gian tới em sẽ tiếp tục nghiên cứu chuyên sâu hơn và cố gắng xây dựng được chương trình kiểm thử tự động dựa vào kỹ thuật kết hợp giữa Logic Hoare với kỹ thuật kiểm thử dựa vào kịch bản dòng dữ liệu.

Trong quá trình làm luận văn, em đã cố gắng rất nhiều, tuy nhiên không tránh khỏi những thiếu sót, em mong rằng sẽ nhận được các ý kiến đóng góp của các Thầy giáo, Cô giáo, các bạn bè, đồng nghiệp để luận văn ngày càng hoàn thiện hơn.

TÀI LIỆU THAM KHẢO

Tiếng Việt:

[1] Phạm Ngọc Hùng, Trương Anh Hoàng và Đặng Văn Hưng (2014), *Giáo trình kiểm thử phần mềm.*

Tiếng Anh:

[2] Bath, G., McKay, J.: Praxiswissen Softwaretest - Test Analystund Technical Test Analyst. Dpunkt, Heidelberg (2010).

[3] Beck (2002), *Test driven development: By example*, Addison Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[4] Fosdick Lloyd D. and Osterweil Leon J. (1976), *Data flow analysis in software reliability*, ACM Comput. Surv. 8, no. 3, 305–330.

[5] Huang J. C. (1979), *Detection of data flow anomaly through program instrumentation*, IEEE Trans. Softw. Eng. 5, no. 3, 226–236

[6] K. B. Gallagher and J. R. Lyle, "Using program slicing in software mainte-nance.," IEEE Trans. Software Eng. , vol. 17, no. 8, pp. 751–761, 1991.

[7] Liggesmeyer (2009), P.: Software-Qualität: Testen, Analysieren und Ve rifizieren von Softwa re, 2nd edn. Spektrum-Akademischer Verlag, Berlin.

[8] M. Weiser, *"Program slicing.,"* IEEE Trans. Software Eng., vol. 10, no. 4, pp. 352–357, 1984.

[9] M. Weiser, *"Program slicing.,"* in IC SE , pp. 439–449, 1981.

[10] Majchrzak, T.A., Kuchen, H.: IHK-Projekt Softwaretests: Auswertung. In: Working Pa pers, Vol. 2. Förderkreis der Angewa ndten Informatik an der Westfälischen Wilhelms-Universität, Münster e .V. (2010).

[11]Michael Kart (2012), *Behavior-driven development: conference tutorial*, J. Comput. Sci.Coll. 27, no. 4, 75–75.

[12] P. C. Jorgensen, *Software Testing: A Craftsman's Approach* . CRC Press, 2nd ed., 2002.

[13] Pezze, M., Young, M.: *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, New York (2007).

[14] Roitzsch, E .H.P.: Analytische Softwarequalitätssicherung in Theorie und Praxis: Der Wegzur Software mit hoher Qualität durch statisches Prüfen, dynamisches Testen, formales Beweisen. Monsensteinund Vannerdat (2005).

[15] S.Liu (2004), *Formal Engineering for Industrial Software Development Using the SOFL Method*. Springer-Verlag, ISBN 3-540-20602-7.

[16] S. Liu and S. Nakajima (2010), *A Decompositional Approach to Automatic Test Case Generation Based on Formal Specifications*. In 4th IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI 2010), pages 147 {155, Singapore, June 9-11 2010. IEEE CS Press}.

[17] S. Liu, T. Hayashi, K. Takahashi, K. Kimura, T. Nakayama, and S. Nakajima (2010), *Automatic Trans-formation from Formal Specifications to Functional Scenario Forms for Automatic Test Case Genneration*. In 9th International Conference on Software Methodologies, Tools and Techniques (SoMet 2010), page to appear, Yokohama city, Japan, Sept. 29- Oct. 1 2010. IOS International Publisher.

[18] S. Rapps and E. J. Weyuker, "*Selecting software test data using data flow information.,"* IEEE Trans. Software Eng. , vol. 11, no. 4, pp. 367–375, 1985.

[19] S.Liu and S.Nakajima (2011), *A "Vibration" method for Automatically Generating Test Cases Based on Formal Specifications*. In 18th Asia-Pacific Software Engineering Conference (APSEC 2011), pages 73{80, HCM City, Vietnam, Dec. 5-8 2011. IEEE CS Press.

[20] Shaoying Liu, *"Utilizing Hoare Logic to Strengthen Testing for Error Detection in Programs"*.

[21] Sneed, H.M., Winter, M.: *Testen Objektorientierter Software*. Hanser, München (2002).

[22] Zeller, A.: Why Programs Fail: *A Guide to Systematic Debugging*. Morgan Kaufmann, San Francisco (2006).